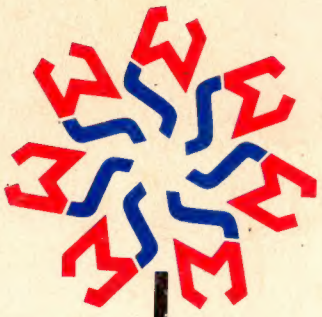


SYMSAC '76

PROCEEDINGS OF THE

1976 ACM SYMPOSIUM ON SYMBOLIC AND ALGEBRAIC COMPUTATION

AUGUST 10-12, 1976
YORKTOWN HEIGHTS, NEW YORK

Edited by R. D. Jenks

R. Fateman

Copyright © 1976 By

The Association for Computing Machinery
1133 Avenue of the Americas
New York, New York 10036

Additional copies of these Proceedings may be ordered prepaid from:

Association for Computing Machinery, Inc.
P.O. Box 12105
Church Street Station
New York, New York 10249

Prices:

Members:	$15.00 prepaid
Non-Members:	$20.00 prepaid

SYMSAC '76

PROCEEDINGS OF THE

1976 ACM SYMPOSIUM ON SYMBOLIC AND ALGEBRAIC COMPUTATION

AUGUST 10-12, 1976
YORKTOWN HEIGHTS, NEW YORK

Edited by R. D. Jenks

R. Fateman

General Chairman

James H. Griesmer

Program Chairman

B. F. Caviness

Program Committee

W. S. Brown
Richard D. Jenks
Alfonso Miola
Joel Moses
David R. Musser
Joachim Neubüser

Proceedings Editor

Richard D. Jenks

Treasurer

H. D. Knoble

We wish to acknowledge the generous financial support which we have received from

The IBM Research Division
The IBM World Trade Europe/Middle East/Africa Corporation
The IBM World Trade Americas/Far East Corporation

Preface

This Symposium marks another in a series of highly successful SIGSAM-sponsored symposia held in the United States at five-year intervals, the previous ones having taken place in 1966 and 1971. It also builds on the success of other meetings that SIGSAM has cosponsored in the recent past, such as the Mathematical Software II Conference and EUROSAM 74. At the same time this Symposium will be the first of a series of annual meetings sponsored by SIGSAM, possibly arranged in conjunction with other organizations, which SIGSAM members can utilize for the presentation of their recent results.

The response to the call for papers for this Symposium has far exceeded the early estimates of Symposium organizers. The Program Committee has scheduled a total of 12 sessions during the three days of the Symposium during which 4 invited and 51 submitted papers will be presented. A fifth invited paper will be given at the symposium banquet, and a panel discussion on *Future Directions in Symbolic Systems*, will be held on a second evening.

It was George Collins, SIGSAM Chairman, who first proposed the idea of holding this Symposium. During the past year, George has been most generous in his support and encouragement in matters relating to the conference. We also wish to acknowledge the secretarial assistance rendered by Barbara Duddleston and Gloria Zalewski at the Illinois Institute of Technology in their handling of the submitted papers sent to the Program Chairman. Because much of the Symposium activity, including publicity, local arrangements, and proceedings, was centered at Yorktown Heights, a considerable secretarial burden was placed on Margaret Casey. She carried out innumerable assignments with her usual efficiency and equanimity. To these individuals, to the referees, to the members of the Program Committee, the Organizing Committee, and the SIGSAM Executive Committee, we wish to express our sincere thanks.

James H. Griesmer
General Chairman

B. F. Caviness
Program Chairman
7/1/76

Referees

The persons listed below have contributed their time and efforts to the reviewing process for this symposium, and their invaluable assistance to the Program Committee is gratefully acknowledged.

S. K. Abdali, RPI
C. M. Anderson, College of William and Mary
A. O. L. Atkin, University of Illinois, Chicago Circle
S. J. Brodsky, Stanford University
S. Cabay, University of Alberta
R. Carroll, University of Utah
B. G. Claybrook, Virginia Polytechnic Institute and State University
G. E. Collins, University of Wisconsin
H. I. Epstein, Boston College
M. Evens, IIT
R. J. Fateman, University of California at Berkeley
S. I. Feldman, Bell Labs
J. Fitch, Cambridge University
W. Fullerton, Los Alamos Scientific Laboratory
M. R. Garey, Bell Labs
M. Griss, University of Utah
J. V. Guttag, USC
A. D. Hall, Jr., Bell Labs
B. K. P. Horn, MIT
E. Horowitz, USC
J. A. Howell, Los Alamos Scientific Laboratory
S. C. Johnson, Bell Labs
H. Jürgensen, Christian-Albrechts Universität, Kiel
B. Kolman, Drexel University
M. J. Levine, Carnegie-Mellon
J. D. Lipson, University of Toronto
R. G. K. Loos, Universität Kaiserslautern
S. S. Magliveras, SUNY Binghamton
M. T. McClellan, University of Maryland
J. McKay, Concordia University
R. T. Moenck, University of Waterloo
M. Newman, National Bureau of Standards
E. W. Ng, JPL
A. C. Norman, Cambridge University
J. R. Pinkert, University of Tennessee
V. Pless, University of Illinois, Circle Campus
R. A. Rink, McMaster University
R. Risch, IBM Research
M. Rosenlicht, University of California at Berkeley
R. Roskies, University of Pittsburgh
S. K. Sahni, University of Minnesota
S. C. Schaller, University of Wisconsin
J. L. Selfridge, Northern Illinois University
C. C. Sims, Rutgers University
A. Sklar, IIT
T. A. Standish, University of California at Irvine
J. C. Thompson, University of Oklahoma
R. G. Tobey, IIT
B. Trager, MIT
H. F. Trotter, Princeton University
P. S. Wang, MIT
B. Wegbreit, Xerox Palo Alto Research Center
M. B. Wells, Los Alamos Scientific Laboratory
K. E. Wendland, IIT
D. Y. Y. Yun, IBM Research
H. J. Zassenhaus, Ohio State University
R. Zippel, MIT

Program

TUESDAY, AUGUST 10

9:00 - 10:50

Welcoming Remarks
R. E. Gomory, Vice President and Director of Research, IBM Corporation

Session 1: Computational Group Theory I
Chairman: J. H. Griesmer, IBM Research

On Determining the Order of a Group (Invited Paper) 1
J. J. Cannon, University of Sydney, and ETH, Zurich

Calculating Presentations for Certain Kinds of Quotient Groups 2
M. F. Newman, Australian National University

Collection 9
G. Havas and T. Nicolson, Australian National University

10:50 - 11:10 Coffee

11:10 - 12:40

Session 2: Algorithms I
Chairman: J. Moses, MIT

Polynomial Real Root Isolation by Differentiation (Invited Paper) 15
G. E. Collins, University of Wisconsin
R. Loos, Universität Kaiserslautern

On Square-Free Decomposition Algorithms. 26
D. Y. Y. Yun, IBM Research

Algorithms for Gaussian Integer Arithmetic 36
B. F. Caviness, IIT, and G. E. Collins, University of Wisconsin

12:40 - 1:40 Lunch

1:40 - 3:40

Session 3: System Design and Implementation
Chairman: S. R. Petrick, IBM Research

A New Reduce Model for Algebraic Simplification 46
A. C. Hearn, University of Utah

The Definition and Use of Data Structures in Reduce 53
M. Griss, University of Utah

A Pattern Compiler 60
R. D. Jenks, IBM Research

A Draft Description of the Group Theory Language Cayley. 66
J. J. Cannon, University of Sydney, and ETH, Zurich

Recursive Programming through Table Look-up 85
D. Friedman, D. Wise, and M. Wand, Indiana University

Symbolic Evaluation of Programs--A Look at Loop Analysis. 90
T. E. Cheatham and J. A. Townley, Harvard University

3:40 - 4:00 Coffee

4:00 - 5:40

Session 4: Applications I
Chairwoman: J. E. Sammet, IBM Federal Systems Division

Qualitative Analysis of Mathematical Expressions Using Computer Symbolic Mathematics 97
D. Stoutemyer, University of Hawaii

The Efficient Calculation of Symbolic Determinants 105
J. Smit, Technische Hogeschool Twente

Automatic Symbolic Solution of Differential Equations of First Order and First Degree 114
P. Schmidt, Universität Bonn

Projective Representations for Energy Band Calculations 126
R. Head, University of Arkansas

Perturbation Calculations for the Spin Up Problem Using Reduce 133
I. Cohen and F. Bark, University of Stockholm

7:30 p.m Symposium Banquet

Banquet Speaker: W. Kahan, University of California, Berkeley
Why Do We Need a Shirtpocket Symbolic Calculator?

WEDNESDAY, AUGUST 11

9:00 - 10:30

Session 5: Polynomial Multiplication
Chairman: B. F. Caviness, IIT

Practical Fast Polynomial Multiplication (Invited Paper) 136
R. T. Moenck, University of Waterloo

Arithmetic Complexity of Unordered Sparse Polynomials 149
F. Gustavson and D. Y. Y. Yun, IBM Research

Hashing Lemmas on Time Complexities with Applications to Formula Manipulation 154
E. Goto and Y. Kanada, University of Tokyo

10:30 - 10:50 Coffee

10:50 - 12:30

Session 6: Computational Group Theory II
Chairman: C. C. Sims, Rutgers University

A Machine Independent Implementation of a Collection Algorithm for the Multiplication of Group Elements 159
V. Felsch, Rhein.-West. Techn. Hochschule Aachen

The Schreier Algorithm for Matrix Groups 167
G. Butler, University of Sydney

CAMAC 171
V. Pless, University of Illinois at Chicago Circle

A Program for Determining Fundamental Units 177
M. Pohst, Universität zu Köln

On the Use of Computer Programs for the Investigation of Kronecker Classes 183
R. Land, Universität zu Köln

12:30 - 1:20 Lunch

11:10 - 12:50

Session 10: Computational Group and Number Theory
Chairman: H. F. Trotter, Princeton University

The Aliquot Project: An Application of Job Chaining in Number Theoretic Computing 276
M. C. Wunderlich, Northern Illinois University

Generalization of Manin's Conditional Algorithms 285
H. Zimmer, Universität des Saarlandes

Computers in the Study of Lie Algebras 300
B. Kolman, Drexel University, and R. Beck, Villanova University

Computational Techniques and the Structure of Groups in a Certain Class 312
C. M. Campbell, University of St. Andrews

The Computation and Application of Vector Congruence Classes 322
Z. L. Lichtman, Tel-Aviv University

12:50 - 1:40 Lunch

1:40 - 3:20

Session 11: Algorithms III
Chairman: A. Miola, IAC, Rome

Simplification of Radical Expressions 329
B. F. Caviness, IIT
R. Fateman, University of California, Berkeley

Canonical Representatives for Residue Classes of a Polynomial Ideal 339
M. Lauer, Universität Kaiserslautern

Taking Advantage of Zero Entries in the Exact Inverse of Sparse Matrices 346
P. Wang and T. Minamikawa, MIT

Completing nth Powers of Polynomials 351
B. Trager, MIT, and D. Y. Y. Yun, IBM Research

A Polynomial Decomposition Algorithm 356
D. A. Barton and R. Zippel, MIT

3:20 - 3:40 Coffee

3:40 - 5:20

Session 12: Applications II
Chairman: G. E. Collins, University of Wisconsin

ASHMEDAI and a Large Algebraic Problem 359
M. J. Levine, Carnegie-Mellon University
R. Roskies, University of Pittsburgh

An Approach to Automatic Asymptotic Expansions 365
R. Fateman, University of California, Berkeley

A Study of Alternative Methods for the Symbolic Calculation of Elliptic Integrals 372
E. W. Ng, Jet Propulsion Lab., and D. Polajnar, USC

Symbolic/Numeric Algorithms for Partial Differential Equations 377
D. Ulery and H. Khalil, University of Delaware

Use of Symbolic and Numeric Methods in an Algorithm for the Approximation of Multivariate Functions 382
D. K. Kahaner and M. Wells, Los Alamos Scientific Lab.

Concluding Remarks
G. E. Collins, University of Wisconsin

1:20 - 3:00

Session 7: Symbolic Systems and Facilities
Chairwoman: J. A. Howell, Los Alamos Scientific Laboratory

Design Features of COBALG 185
J. Fitch, P. Herbert, A. Norman, Cambridge University

Basic Algorithms in FORMAC: Design and Verification 189
K. Bahr, GMD/IFV, Darmstadt

Univariate Power Series Expansions in Algebraic Manipulation 198
R. Zippel, MIT

The MACSYMA "Big-Floating-Point" Arithmetic System 209
R. Fateman, University of California, Berkeley

Interval Arithmetic Applied to Polynomial Remainder Sequences 214
J. Pinkert, University of Tennessee

3:00 - 3:20 Coffee

3:20 - 5:00

Session 8: Algorithms and Foundations
Chairman: J. Hanson, Cleveland State University

Algebraic Factoring and Rational Function Integration 219
B. Trager, MIT

Some Problems about Polynomials 227
M. Mignotte, Université de Strasbourg

Using Basis Computation to Determine Pseudo-Multiplicative Independence 229
H. I. Epstein, Boston College

On the Sign of a Real Algebraic Number 238
S. Rump, Universität Kaiserslautern

Algorithms for Symmetrical Polynomials 242
E. Lauer, Universität Kaiserslautern

8:00 p.m. - 10:30 p.m.

Panel Discussion: Future Directions in Symbolic Systems

R. D. Jenks, IBM Research, Moderator
J. J. Cannon, University of Sydney, and ETH, Zurich
W. M. Gentleman, University of Waterloo
A. C. Hearn, University of Utah
R. Loos, Universität Kaiserslautern
J. Moses, MIT

THURSDAY, AUGUST 12

9:00 - 10:50

Session 9: Algorithms II
Chairman: Carl Engelman, MITRE Corporation

Algebraic Algorithms using p-adic Constructions (Invited Paper) 248
D. Y. Y. Yun, IBM Research

Newton's Method: A Great Algebraic Algorithm 260
J. Lipson, University of Toronto

On the Subresultant PRS Algorithm 271
W. S. Brown, Bell Laboratories

Polynomial Real Root Isolation Using Descarte's Rule of Signs 272
G. E. Collins and A. G. Akritas, University of Wisconsin

10:50 - 11:10 Coffee

Author Index

Akritas 272
Bahr 189
Bark 133
Barton 356
Beck 300
Brown 271
Butler 167
Campbell 312
Cannon 1,66
Caviness 36,329
Cheatham 90
Cohen 133
Collins 15,36,272
Epstein 229
Fateman 209,329,365
Felsch 159
Fitch 185
Friedman 85
Goto 154
Griss 53
Gustavson 149
Havas 9
Head 126
Hearn 46
Herbert 185
Jenks 60
Kahaner 382
Kanada 154
Khalil 377
Kolman 300
Land 183
Lauer, E. 242
Lauer, M. 339
Levine 359
Lichtman 322
Lipson 260
Loos 15
Mignotte 227
Minamikawa 346
Moenck 136
Newman 2
Ng 372
Nicolson 9
Norman 185
Pinkert 214
Pless 171
Pohst 177
Roskies 359
Rump 238
Schmidt 114
Smit 105
Stoutemyer 97
Townley 90
Trager 219,351
Ulery 377
Wand 85
Wang 346
Wells 382
Wise 85
Wunderlich 276
Yun 26,149,248,351
Zimmer 285
Zippel 198,356

On Determining the Order of a Group

John J. Cannon
Department of Pure Mathematics
University of Sydney
Sydney
and
Forschungsinstitut für Mathematik
Eidgenössische Technische Hochschule
Zurich

The recent past has seen a proliferation of algorithms designed to carry out a variety of calculations with groups. Chief among these is a family of algorithms for determining the order of a group. While computing the order of a group is often of great interest in itself, the importance of many of these algorithms lies in the fact that they also "construct" the group in such a way that makes possible the application of powerful algorithms for computing aspects of the group's structure.

In the first half of the paper I treat algorithms for groups given by concrete sets of generators. Commencing with an efficient algorithm for explicitly listing all the elements of a small group, I then describe Sims' Schreier technique for permutation groups and Butler's extension of it to matrix groups. Algorithms for working with groups given by abstract defining relations comprise the second half of the paper. A brief outline of the Todd-Coxeter algorithm and some of its applications is given. This is followed by an account of the recently developed algorithm of Macdonald and Wamsley for nilpotent groups. The paper concludes with some examples showing how certain problems may be successfully attacked using a combination of methods.

Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation

Calculating Presentations for Certain Kinds of Quotient Groups

M.F. Newman

Department of Mathematics,
Institute of Advanced Studies,
Australian National University,
Canberra, Australia.

Abstract. Over the last few years a new algorithm has emerged for manipulation with finite presentations of groups. It enables one to calculate from a given finite presentation a useful presentation for certain nilpotent quotient groups of the group presented by the input. The algorithm has been machine implemented. With the aid of these machine implementations it has been possible to dig more deeply into the structure of some groups, in particular finite groups of prime-power order, and the prospects for further investigations seem bright.

In this paper I am going to describe an algorithm of fairly recent origin for calculating useful presentations for certain (nilpotent) quotient groups of a group given in terms of generators and defining relations. This algorithm adds significantly to the armoury of techniques available for investigating groups. The algorithm is well suited for machine implementation. This has been done to varying extents by Macdonald (1974), Bayes, Kautsky and Wamsley (1974), W. Felsch (1974), Alford, Havas and Newman (1975) and Havas and Newman. With the aid of these implementations progress has been made on a number of questions, some of quite long-standing.

Among the more spectacular results which have been obtained are some related to the so-called Burnside Problem. This is really a number of related problems which arise out of some questions recorded by Burnside (1902) and attempts to answer them. This is not the occasion for going into the detail of that, but rather one for going directly to the relevant parts. For positive integers d, k there is among the d-generator groups of exponent k (every element has order dividing k) one, $B(d, k)$, of which all the others are homomorphic images. It has been known since Burnside's paper that $B(2, 4)$ is finite of order at most 2^{12} and since Sanov's paper of 1940 that $B(d, 4)$ is finite for all d though the order bound his proof gives is astronomical. Determining the order of these groups has remained difficult. Tobin (1954) proved that $B(2, 4)$ has order 2^{12}. Using the algorithm to be described (with some additional features) Bayes, Kautsky and Wamsley (1974) proved that $B(3, 4)$ has order 2^{69} and Alford, Havas and Newman (1975) that $B(4, 4)$ has order 2^{422}. Kostrikin (1959) proved that among the *finite* d-generator groups of prime exponent p there is one, $K(d, p)$, of which all the others are homomorphic images. Wamsley (see Havas, Wall and Wamsley (1974)) has used the algorithm to show that the order of $K(2, 5)$ is 5^{34}. Macdonald (1974, p. 112) used the algorithm to settle the discussion between Coxeter-Moser (1957, 1965, 1972) and M. Hall (1958, 1973) about the number of fourth powers needed to define $B(2, 4)$ by showing that every presentation of $B(2, 4)$ on two generators needs at least nine relations. Other results obtained with the aid of the algorithm occur in some of the papers cited. The algorithm has acted as a catalyst in proving that certain groups are infinite (Newman 1976a). It would seem to have a useful role to play in the determination of groups of prime-power order (Newman 1976b).

The description of the algorithm to be given uses descending central chains of subgroups of groups and presentations of groups in terms of generators and relations. The appropriate definitions will be sketched both to serve as a reminder and to establish notation. If A, B are subsets of a group G, then $[A, B]$ will denote the subgroup of G generated by all commutators $[a, b] = a^{-1}b^{-1}ab$ with a in A and b in B. A chain

$$G_0 \geq G_1 \geq \ldots \geq G_c \geq G_{c+1} \geq \ldots$$

of subgroups of a group G is a *descending central chain* of subgroups of G if $G_0 = G$ and $G_{c+1} \geq [G_c, G]$. The *lower central chain* of G is (standardly but in some respects awkwardly) defined by $\gamma_0(G) = \gamma_1(G) = G$ and $\gamma_{c+1}(G) = [\gamma_c(G), G]$ for $c \geq 1$. A group is *nilpotent* if some $\gamma_{c+1}(G)$ is the identity subgroup E; if $\gamma_c(G) \neq E$ it has nilpotency class c. In particular every abelian (commutative) group is nilpotent of class 1. In this context we will be concerned only with descending central chains

$$G = \eta_0(G) \geq \ldots \geq \eta_c(G) \geq \eta_{c+1}(G) \geq \ldots$$

which can be described, like the lower central chain, in a *uniform* way for all groups (technically for each group G each $\eta_c(G)$ is to be verbal in G). If $\eta_c(G) = E$ and $\eta_{c-1}(G) \neq E$ then G has η-class c; a group with η-class c has nilpot-

ency class at most c. Other examples of uniform descending central chains are, one for each prime p, the *lower exponent-p-central* chains

$$G = P_0(G) \geq \ldots \geq P_c(G) \geq P_{c+1}(G) \geq \ldots$$

where $P_{c+1}(G) = [P_c(G), G](P_c(G))^p$. These are the chains involved in the implementation of the algorithm by Havas and Newman which will be referred to as the Canberra implementation (a program is available from them). The Macdonald (1974), Bayes, Kautsky and Wamsley (1974) and W. Felsch (1974) implementations use a different set of uniform descending central chains. The lower central chain has been used in some hand calculations; there is no obstacle in principle to its use in a machine implementation.

A *presentation* of a group G in terms of generators and relations consists of a set B (of generators) and a set R of relations $u = v$ where u, v are words in B such that G is isomorphic to the quotient group F/R where F is the free group freely generated by B and R is the normal subgroup of F generated, as normal subgroup, by all the uv^{-1} with $u = v$ in R. For example

(i) $B = \{b_1, b_2, b_3\}$
$R = \{b_1b_2 = b_3, b_2b_3 = b_1, b_3b_1 = b_2\}$,

(ii) $B = \{b_1, b_2\}$
$R = \left\{b_1^2 = \emptyset, b_2^2 = \emptyset\right\}$

where $\emptyset$ is the empty word. As these examples show conventional abbreviations such as b_1^2 will be used in this discussion and are to some extent used in implementations; nevertheless words are always regarded as words in the free group sense, that is words in $\{b_1, \ldots, b_n\}$ are strings of $b_1, \ldots, b_n, b_1^{-1}, \ldots, b_n^{-1}$ (not necessarily reduced). The presentation will be denoted $\langle B; R \rangle$ and in particular cases standard and obvious abbreviations will be used, so example (ii) becomes

$$\left\langle b_1, b_2; b_1^2 = b_2^2 = \emptyset \right\rangle .$$

Each of the sample presentations is *finite* in the sense that both B and R are finite. All presentations which arise in this context are finite.

The algorithm accepts as input a finite presentation $\langle B; R \rangle$ and a finite uniform descending central chain $\eta_0, \eta_1, \ldots, \eta_{\mathrm{MAXCL}}$ and produces as output a special type of presentation for the nilpotent quotient $G/\eta_{\mathrm{MAXCL}}(G)$ where G is the group presented by $\langle B; R \rangle$. It is therefore called the *nilpotent quotient algorithm*. The output presentations of the nilpotent quotient algorithm will be described in the next few paragraphs. Their most important feature is that they give a fairly compact description of the group in question from which the elements and their products can be comparatively easily extracted. Such presentations in the context of groups of prime-power order have a long, though somewhat low-key, history going back at least to a noted paper of Sylow (1872). As an example, if the input is the presentation $\left\langle b_1, b_2; b_1^2 = b_2^2 = \emptyset \right\rangle$ and the lower exponent-2-central chain $P_0, \ldots, P_c$ $(c \geq 3)$, then the output presentation is

$$\Big\langle a_1, \ldots, a_{c+1}; a_1^2 = \emptyset,\ a_2^2 = \emptyset,$$
$$a_i^2 = a_{i+1}a_{i+2}, \qquad 3 \leq i \leq c-1,$$
$$a_c^2 = a_{c+1},\ a_{c+1}^2 = \emptyset,$$
$$[a_j, a_i] = a_{j+1}, \qquad 2 \leq j \leq c,\ 1 \leq i \leq 2,\ i \neq j,$$
$$[a_j, a_i] = \emptyset, \qquad \text{otherwise}\Big\rangle$$

with a_1, a_2 corresponding to b_1P_c, b_2P_c. Call this presentation $\mathcal{P}_c$.

The initial discussion of output presentations will encompass a wider class of presentations than the present context requires. This can be done without cost and enables related work such as that reported by V. Felsch in these proceedings and by Wamsley (1976) to be brought under the umbrella of the discussion. Consider to begin with presentations of the form

$$\Big\langle a_1, \ldots, a_n; a_i^{\rho(i)} = u_i,\ 1 \leq i \leq n,$$
$$a_ja_i = a_iv_{ij},\ 1 \leq i < j \leq n,$$
$$a_ja_i^{-1} = a_i^{-1}w_{ij},$$
$$1 \leq i < j \leq n,\ \rho(i) = 0\Big\rangle$$

where each $\rho(i)$ is a non-negative integer, u_i, v_{ij}, w_{ij} are words in $\{a_{i+1}, \ldots, a_n\}$ and $u_i = \emptyset$ if $\rho(i) = 0$. Such presentations will be called *power-commutation* presentations. The terminology is still in a state of flux. Jürgensen (1970) used the term *AG*-system; Felsch uses power-commutator presentation. (This term will be used here later for a more restricted class of presentations.) The group described by a power-commutation presentation is polycyclic (soluble and satisfies the maximal condition for subgroups) and, conversely, every polycyclic group has a power-commutation presentation. The vector $(\rho(1), \ldots, \rho(n))$ will be called the *exponent vector* of the presentation. It is not difficult to prove that in groups with a power-commutation presentation every element can be written in the form

$$\prod_{k=1}^{n} a_k^{\xi(k)} \quad \text{with} \quad 0 \leq \xi(k) < \rho(k) \quad \text{or} \quad \rho(k) = 0 .$$

Call words of this form *normal* (Felsch uses normed). Thus every word in $\{a_1, \ldots, a_n\}$ is equivalent to a normal word.

There are algorithms for calculating for every word a normal word equivalent to it. If a word is not normal it has a subword in the following list:

LIST A

(a) $a_i^{-1}a_i$ $\quad 1 \le i \le n$,

(b) $a_i a_i^{-1}$ $\quad 1 \le i \le n$,

(c) $a_j a_i$ $\quad 1 \le i < j \le n$,

(d) $a_i^{\rho(i)}$ $\quad 1 \le i \le n,\ \rho(i) > 0$,

(e) $a_j a_i^{-1}$ $\quad 1 \le i < j \le n,\ \rho(i) = 0$,

(f) a_i^{-1} $\quad 1 \le i \le n,\ \rho(i) > 0$,

(g) $a_j^{-1}a_i$ $\quad 1 \le i < j \le n,\ \rho(j) = 0$,

(h) $a_j^{-1}a_i^{-1}$ $\quad 1 \le i < j \le n,\ \rho(i) = \rho(j) = 0$.

It follows from the group property and the relations of the presentation that these words are equivalent, respectively, to the following words

LIST B

(a) $\emptyset$

(b) $\emptyset$

(c) $a_i v_{ij}$

(d) u_i

(e) $a_i^{-1} w_{ij}$

(f) $a_i^{\rho(i)-1} u_i^{-1}$

(g) $a_i v_{ij}^{-1}$

(h) $a_i^{-1} w_{ij}^{-1}$

The following type of procedure applied to a word can be shown to always terminate

1. if w is normal, stop;
2. $w \leftarrow$ the result of replacing a subword of w which occurs in list A by its equivalent in list B, and return to 1.

Moreover there are bounds, in terms of suitable parameters of the presentation and the length of the initial word, for the number of steps to termination and the lengths of the intermediate words. Such procedures will be called *collection processes*.

EXAMPLE. Consider the power-commutation presentation $\mathcal{D}_c$ and the word $a_2a_1a_1$; this can be collected either

$$a_2\underline{a_1a_1} = a_2$$

or

$$\begin{aligned} \underline{a_2a_1}a_1 &= a_1a_2\underline{a_3a_1} \\ &= a_1\underline{a_2a_1}a_3a_4 \\ &= a_1a_1a_2a_3a_3a_4 \end{aligned}$$

when there is a choice again, however all choices lead after another $2c - 3$ steps to a_2

An efficient collection process is one important part of an efficient nilpotent quotient algorithm. The Canberra implementation of the nilpotent quotient algorithm follows Wamsley's lead in using collection *from the right* which is the collection process in which the subword replaced in step 2 is always the right-most. (Wamsley's strong advocacy of this collection is hard to document; see Sag and Wamsley (1973, p. 365) and Bayes, Kautsky and Wamsley (1974, p. 83) for scant traces.) The collection in the Canberra implementation of the nilpotent quotient algorithm is described in the paper by Havas and Nicholson in these proceedings; it incorporates as a significant feature Nicholson's combinatorial collection.

There are power-commutation presentations in which two different normal words represent the same element of the corresponding group. For example

$$\langle a_1, a_2, a_3;\ a_1^2 = a_2,\ a_2^2 = a_3,\ a_3^2 = \emptyset,$$
$$a_2a_1 = a_1a_2a_3,\ a_3a_1 = a_1a_3,\ a_3a_2 = a_2a_3 \rangle .$$

For this presentation a_1a_2, $a_1a_2a_3$ represent the same element because

$$a_1a_2 = a_1a_1^2 = a_1^2a_1 = a_2a_1 = a_1a_2a_3 .$$

This situation arises naturally in the nilpotent quotient algorithm. A power-commutation presentation in which every element is represented by exactly one normal word will be called *consistent* (following Wamsley (1974) who originated the approach presented here). For a consistent power-commutation presentation the set of elements of the group can be regarded as the set of normal words and the multiplication is defined by collection, that is, the product of two normal words is the normal word which results from collecting their concatenation. The order of the group is therefore the number of normal words; so, if all the $\rho(i)$ are positive, the group has finite order $\rho(1) \ldots \rho(n)$. There is a finite test for consistency which will be described below. Moreover, if the test shows inconsistency, the result of the test can be used to construct another power-commutation presentation for the same group whose exponent vector properly divides that of the original presentation; that is, each component divides and at least one component properly divides the corresponding component. The test can then be applied to this new presentation. Since the divisibility order of exponent vectors is a well-order, this process will terminate after a finite number of steps in a consistent power-commutation presentation for the group under consideration. The list of words below (List C) provides a basis for testing the consistency of a power-commutation presentation. The list consists of certain words which have essentially two subwords from list A above.

The test involves applying two sufficiently different collection processes to each of these test words. One of these can be collection from the right, the other can be a collection which for each test word replaces the left subword from list A and then proceeds by collection from the right. If each test word collects to the same normal word under both collections, then the presentation is consistent (the proof requires some work; a paper containing it is being prepared). For example the presentation at the beginning of this paragraph can be seen to be inconsistent on the word a_1^3 for

$$a_1^3 = a_1\underline{a_1^2} = a_1a_2 \quad \text{and} \quad a_1^3 = \underline{a_1^2}a_1 = \underline{a_2a_1} = a_1a_2a_3 .$$

LIST C

(a) $a_k a_j a_i$ $\quad i < j < k$,

(b) $a_k a_j a_i^{-1}$ $\quad i < j < k$, $\rho(i) = 0$,

(c) $a_j a_i a_i^{-1}$ $\quad i < j$, $\rho(i) = 0$,

(d) $a_j a_i^{-1} a_i$ $\quad i < j$, $\rho(i) = 0$,

(e) $a_j^{-1} a_j a_i^{-1}$ $\quad i < j$, $\rho(i) = 0$,

(f) $a_j a_i^{\rho(i)}$ $\quad i < j$, $\rho(i) > 0$,

(g) $a_j^{\rho(j)} a_i$ $\quad i < j$, $\rho(j) > 0$,

(h) $a_j^{\rho(j)} a_i^{-1}$ $\quad i < j$, $\rho(i) = 0$, $\rho(j) > 0$.

(i) $a_i^{\rho(i)+1}$ $\quad \rho(i) > 0$.

From this one deduces that the equation $a_3 = \emptyset$ must hold. It follows that

$$\langle a_1, a_2; a_1^2 = a_2, a_2^2 = \emptyset, a_2 a_1 = a_1 a_2 \rangle$$

presents the same group. The consistency test shows that this presentation is consistent, so the group has order 4 . (Of course this is easy to see directly because the cyclic group of order 4 has such a presentation.)

The output presentations of the nilpotent quotient algorithm have the further property that the v_{ij}, w_{ij} can be written $a_j v_{ij}^*$, $a_j w_{ij}^*$ respectively where v_{ij}^*, w_{ij}^* are normal words in $\{a_{j+1}, \ldots, a_n\}$. Such presentations may be written

$$\left\langle a_1, \ldots, a_n; \begin{array}{l} a_i^{\rho(i)} = u_i, \quad 1 \le i \le n, \\ [a_j, a_i] = v_{ij}^*, \quad 1 \le i < j \le n, \\ [a_j, a_i^{-1}] = w_{ij}^*, \quad 1 \le i < j \le n, \ \rho(i) = 0 \end{array} \right\rangle$$

and will be called power-*commutator* presentations. Every group described by a power-commutator presentation is nilpotent and every finitely generated nilpotent group has such a presentation.

It is now possible to describe the main features of the nilpotent quotient algorithm. The nilpotent quotient algorithm works a class at a time. At the end of the c-th stage it has calculated a consistent power-commutator presentation for $G/\eta_c(G)$ which is an extension (in an obvious sense) of the consistent power-commutator presentation for $G/\eta_{c-1}(G)$. In the example where the input is $\langle b_1, b_2; b_1^2 = b_2^2 = \emptyset \rangle$ for $c \ge 3$ the presentation $\mathcal{P}_c$ for $G/P_c(G)$ has been obtained from that of $G/P_{c-1}(G)$, namely $\mathcal{P}_{c-1}$, by adding the generator a_{c+1} , adding relations which make a_{c+1} central and of order 2 , and modifying the relations

$$a_{c-1}^2 = a_c, \ a_c^2 = \emptyset, \ [a_c, a_1] = \emptyset, \ [a_c, a_2] = \emptyset$$

to

$$a_{c-1}^2 = a_c a_{c+1}, \ a_c^2 = a_{c+1},$$

$$[a_c, a_1] = a_{c+1}, \ [a_c, a_2] = a_{c+1} .$$

To ease the description attention will be restricted to the case in which the allowed descending central chains are limited to finite lower exponent-p-central chains (as in the Canberra implementation). These chains are specified by the two parameters MAXCL and p . The first stage of the nilpotent quotient algorithm calculates a consistent power-commutator presentation for $G/P_1(G)$, the largest abelian-of-exponent-p quotient group of G . This is done in two steps. In the first step the given presentation $\langle B; R \rangle$ is used to set up a homogeneous system of equations over the field of integers modulo p . The equations are calculated from the relations by abelianizing the relations and taking exponents modulo p (and writing the result additively).

EXAMPLE. If the input presentation is

$$\langle b_1, \ldots, b_6; b_1 b_2 = b_3, b_2 b_3 = b_4, b_3 b_4 = b_5, b_4 b_5 = b_6, b_5 b_6 = b_1, b_6 b_1 = b_2 \rangle$$

and $p = 2$, the equations are

$$b_1 + b_2 = b_3, \ b_2 + b_3 = b_4, \ b_3 + b_4 = b_5 ,$$
$$b_4 + b_5 = b_6, \ b_5 + b_6 = b_1, \ b_1 + b_6 = b_2 .$$

In the second step this system of equations is solved (by row-echelonizing). In the example the solution takes the form

$$b_3 = b_1 + b_2, \ b_4 = b_1, \ b_5 = b_2, \ b_6 = b_1 + b_2 .$$

Thus the solution space is two-dimensional and the output of step 1 can be written

$$\langle a_1, a_2; a_1^2 = \emptyset, \ a_2^2 = \emptyset, \ [a_2, a_1] = \emptyset \rangle .$$

Moreover modulo $P_1(G)$ (and abusing language)

$$b_1 = a_1, \ b_2 = a_2, \ b_3 = a_1 a_2 ,$$
$$b_4 = a_1, \ b_5 = a_2, \ b_6 = a_1 a_2 .$$

In general let d be the dimension of the solution space of the system of equations, then the output can be written

$$\left\langle a_1, \ldots, a_d; \begin{array}{l} a_i^p = \emptyset, \ 1 \le i \le d, \\ [a_j, a_i] = \emptyset, \ 1 \le i < j \le d \end{array} \right\rangle$$

and there will be modulo $P_1(G)$ (and with similar abuse of language) expressions for the b_i in terms of the a_j . If $d = 0$, then the only finite p-group quotient group of G is the identity group. If $d = 1$, the finite p-group quotient groups of G are cyclic and can be completely determined by abelianizing the given presentation of G . Thus the nilpotent quotient algorithm is continued only if $d \ge 2$.

The stages of the nilpotent quotient algorithm other than the first all have the same form. The c-th stage $(c \ge 2)$ calculates a consistent power-commutator presentation for $G/P_c(G)$ and may be divided into three steps. The first of these involves taking the output of the $(c-1)$-th stage and using it as a basis for writing down a, usually inconsistent, power-commutator presentation

for the largest (which does exist) d-generator group H of P-class c which has $H/P_{c-1}(H)$ isomorphic to $G/P_{c-1}(G)$. Recall that the output of the $(c-1)$-th stage is a consistent power-commutator presentation, $\langle a_1, \ldots, a_n; \ldots \rangle$ say, for $G/P_{c-1}(G)$. There are two features of output presentations which have not yet been mentioned. Each generator a_i of the presentation can have a weight $\omega(a_i)$ assigned to it which is the stage at which it was added to the presentation and each of the $n - d$ generators of weight at least 2 is "defined" by one of the relations, that is, for each such generator a_i there is a relation of the form $s = ta_i$ where s is a commutator or a p-th power, t is a normal word and both s, t are words in $\{a_1, \ldots, a_{i-1}\}$. These definitions can be used as the basis of Tietze transformations to yield presentations of the form $\langle a_1, \ldots, a_d; S \rangle$ for $G/P_{c-1}(G)$. Let F be the free group on $\{a_1, \ldots, a_d\}$ and S the normal subgroup defined by S, then H is $F/[F, S]S^p$. There are many ways of writing down a power-commutator presentation for H which extends the available consistent power-commutator presentation for $G/P_{c-1}(G)$. If there are q relations which are not definitions, then a simple scheme for this is

1. add one generator a_{n+r} $(1 \le r \le q)$ for each relation which is not a definition;
2. add relations to make these new generators central and of order p;
3. modify the relations which are not definitions: the r-th such relation $s = t$ becomes $s = ta_{n+r}$ (observe this is the definition of a_{n+r}).

however this is usually wasteful of space and effort because it adds many generators which have later to be eliminated. It is not difficult to see that it is unnecessary to add a generator for a relation $[a_j, a_i] = \emptyset$ when $\omega(a_j) + \omega(a_i)$ exceeds c. It is possible to reduce further the number of new generators added at the cost of complicating the procedure for modifying relations. This is done in the Canberra implementation in which one new generator is added for each relation $[a_j, a_i] = \emptyset$ with $\omega(a_j) = c - 1$, $\omega(a_i) = 1$ and one for each relation $a_j^p = \emptyset$ with $\omega(a_j) = c - 1$ and a_j itself defined as a p-th power. The modification of the relations is too lengthy to explain here. It does, however, involve the selective addition of some generators which are guaranteed to be eliminated in the later equation solving; these are called *pseudo-generators*. In the c-th stage of the nilpotent quotient algorithm applied to $\langle b_1, b_2; b_1^2 = b_2^2 = \emptyset \rangle$ the Canberra implementation would add 3 generators and $2c - 4$ pseudo-generators; this compares with $2 + c(c-1)/2$ generators for the simple scheme above and about half that number when weights are taken into account. The work on $B(4, 4)$ would not have been possible without such reductions. Whatever method is used the resulting presentation is generally not consistent. The second step of the c-th stage consists of changing the presentation for H to a consistent power-commutator presentation for H. This involves forming a homogeneous system of equations over the field of integers modulo p. Each equation is obtained by collecting each of the relevant test words in the two ways indicated earlier, equating the resulting normal words and reducing the resulting relation as much as possible. Because the presentation for $G/P_{c-1}(G)$ is consistent, the final relation only involves the new generators and pseudo-generators and so can be interpreted as an equation of the required kind. This system of consistency equations is then solved and the solution used to eliminate new generators and pseudo-generators. The resulting presentation is a consistent power-commutator presentation for H. This presentation may still contain pseudo-generators; they are eliminated in the next step by the relations of G. If the equation system has no non-trivial solution, then $G/P_{c-1}(G)$ has no proper P-class c extention and $G/P_{c-1}(G)$ is the maximal finite p-group quotient of G. In the example of $\langle b_1, b_2; b_1^2 = b_2^2 = \emptyset \rangle$ the presentation for the corresponding H has one generator $a_{c+1} = [a_c, a_1]$ and two pseudo-generators $a_{c+2} = a_1^2$, $a_{c+3} = a_2^2$. If G is $B(2, 4)$ and $c = 5$, the corresponding H has order 2^{21} and so at least nine relations are needed for any two-generator presentation of $B(2, 4)$. The third and final step of the c-th stage incorporates the defining relations for G into the consistent power-commutator presentation for H to yield a consistent power-commutator presentation for $G/P_c(G)$. This (possibly) involves adding further pseudo-generators, one for each generator of the original generating set B which was eliminated in the first stage. The relations with the original generators replaced by the appropriate normal word in the a's are collected to yield a homogeneous system of equations over the integers modulo p which is then solved. The solution leads to the elimination of all the remaining pseudo-generators and possibly some of the new generators.

EXAMPLES. In the running example the relations $b_1^2 = b_2^2 = \emptyset$ lead to $a_{c+2} = a_{c+3} = \emptyset$ and the output presentation $\mathcal{D}_c$.

An example where a pseudo-generator is added in the third step is provided by the presentation

$$\langle b_1, b_2, b_3; b_1b_2 = b_3, b_2b_3 = b_1, b_3b_1 = b_2 \rangle$$

with $p = 2$. In this case $d = 2$ so b_3 is eliminated in the first stage. After the first two steps of the second stage one has

$$\langle a_1, \ldots, a_5; a_1^2 = a_4, a_2^2 = a_5, a_3^2 = a_4^2 = a_5^2 = \emptyset, [a_2, a_1] = a_3, [a_j, a_i] = \emptyset \text{ otherwise} \rangle .$$

The third step consists of adding a_6, replacing b_1, b_2, b_3 in the relations by $a_1, a_2, a_1a_2a_6$ respectively and collecting the relations to get the equations

$$a_6 = \emptyset,\ a_3a_5a_6 = \emptyset,\ a_3a_4a_6 = \emptyset\ .$$

The output presentation is

$$\langle a_1, a_2, a_3;\ a_1^2 = a_2^2 = a_3,\ a_3^2 = \emptyset,$$
$$[a_2, a_1] = a_3,\ [a_3, a_1] = [a_3, a_2] = \emptyset\rangle\ .$$

Hence the group has the quaternion group as a quotient group. It follows that this is the largest finite 2-group quotient of the given group (by taking the nilpotent quotient algorithm a stage further).

It may happen that at the end of the third step all the new generators have been eliminated in which case $G/P_{c-1}(G)$ is the largest finite p-group quotient group of G and no further information can be obtained from this input, so the algorithm is terminated. Otherwise the nilpotent quotient algorithm goes on to the next stage until c = MAXCL when it terminates.

If the nilpotent quotient algorithm terminates because in it c has reached MAXCL, then the group defined by the input presentation may well have larger nilpotent quotient groups than that presented by the output presentation. These could be investigated by extending the central chain. However this process need never end as there are finitely presented groups which have nilpotent quotients of all classes. If the nilpotent quotient algorithm terminates in any other way, then the input group has a largest quotient relative to the given central chain and all extensions of it. However further information in the form of a suitable nilpotency proof is needed before a statement can be made about the input group itself. For example the nilpotent quotient algorithm applied with a lower exponent-2-central chain with MAXCL = 6 to the presentation

$$\langle b_1, b_2;\ b_1^4 = b_2^4 = (b_1b_2)^4 = \left(b_1^{-1}b_2\right)^4 = \left(b_1b_2^2\right)^4$$
$$= \left(b_1^2b_2\right)^4 = \left(b_1^2b_2^2\right)^4 = \left(b_1^{-1}b_2^{-1}b_1b_2\right)^4 = \left(b_1^{-1}b_2b_1b_2\right)^4\rangle$$

shows that this group has a largest finite 2-group quotient group of order 2^{12}. It needs a finiteness proof of some kind to show the group itself is a finite 2-group. It then follows that the group has order 2^{12}

REFERENCES

William A. Alford, George Havas and M.F. Newman (1975), "Groups of exponent four", *Notices Amer. Math. Soc.* 22, A-301.

A.J. Bayes, J. Kautsky and J.W. Wamsley (1974), "Computation in nilpotent groups (application)", *Proc. Second Internat. Conf. Theory of Groups* (Canberra, 1973), 82-89 (Lecture Notes in Mathematics, 372. Springer-Verlag, Berlin, Heidelberg, New York).

W. Burnside (1902), "On an unsettled question in the theory of discontinuous groups", *Quart. J. Pure and Appl. Math.* 33, 230-238.

H.S.M. Coxeter and W.O.J. Moser (1957), *Generators and Relations for Discrete Groups* (Ergebnisse der Mathematik und ihrer Grenzgebiete, 14. Springer-Verlag, Berlin, Göttingen, Heidelberg).

H.S.M. Coxeter and W.O.J. Moser (1965), *Generators and Relations for Discrete Groups* (second edition).

H.S.M. Coxeter and W.O.J. Moser (1972), *Generators and Relations for Discrete Groups* (third edition).

Waltraud Felsch (1974), "Ein commutator collecting Algorithmus zur Bestimmung einer Zentralreihe einer endlichen p-Gruppe", Thesis, RWTH Aachen.

Marshall Hall, Jr. (1958), "Solution of the Burnside problem for exponent six", *Illinois J. Math.* 2, 764-786.

Marshall Hall, Jr. (1958), Review of: *Generators and Relations for Discrete Groups* by H.S.M. Coxeter and W.O.J. Moser (Ergebnisse der Mathematik und ihrer Grenzgebiete, 14. Springer-Verlag, Berlin, Göttingen, Heidelberg, 1957). *Bull. Amer. Math. Soc.* 64, 106-108.

Marshall Hall, Jr. (1973), "Notes on groups of exponent four", *Conf. on Group Theory*, University of Wisconsin-Parkside (1972), 91-118 (Lecture Notes in Mathematics, 319. Springer-Verlag, Berlin, Heidelberg, New York).

George Havas, G.E. Wall, and J.W. Wamsley (1974), "The two generator restricted Burnside group of exponent 5 ", *Bull. Austral. Math. Soc.* 10, 459-470.

H. Jürgensen (1970), "Calculation with the elements of a finite group given by generators and defining relations", *Computational Problems in Abstract Algebra* (Proc. Conf. Oxford, 1967, 47-57. Pergamon, Oxford).

А.И. Кострикин [A.I. Kostrikin] (1959), "О проблеме Бернсайда" [The Burnside problem], *Izv. Akad. Nauk SSSR Ser. Mat.* 23, 3-34; *Amer. Math. Soc. Transl.* (2) 36 (1964), 63-99.

I.D. Macdonald (1974), "A computer application to finite p-groups", *J. Austral. Math. Soc.* 17, 102-112.

M.F. Newman (1976a), "A computer aided study of a group defined by fourth powers", *Bull. Austral. Math. Soc.* 14, 293-297.

M.F. Newman (1976b), "Determinanion of groups of prime-power order", *Proc. Miniconf. Group Theory* (Canberra, 1975) (Lecture Notes in Mathematics, to appear).

T.W. Sag and J.W. Wamsley (1973), "On computing the minimal number of defining relations for finite groups", *Math. Comp.* 27, 361-368.

И.Н. Санов [I.N. Sanov] (1940), "Решение проблемы Бернсайда для показателя 4 " [Solution of Burnside's problem for exponent four], *Leningrad. Gos. Univ. Ped. Inst. Uc. Zap. Mat. Ser.* 10, 166-170.

L. Sylow (1872), "Théorèmes sur les groupes de substitutions", *Math. Ann.* 5, 584-594.

John Joseph Tobin (1954), "On groups with exponent 4 ", PhD Thesis, University of Manchester.

J.W. Wamsley (1974), "Computation in nilpotent groups (theory)", *Proc. Second Internat. Conf. Theory of Groups* (Canberra, 1973), 691-700 (Lecture Notes in Mathematics, 372. Springer-Verlag, Berlin, Heidelberg, New York).

J.W. Wamsley (1976), "Computing soluble groups", *Proc. Miniconf. Group Theory* (Canberra, 1975) (Lecture Notes in Mathematics, to appear).

Collection

George Havas and Tim Nicholson

Department of Mathematics,
Institute of Advanced Studies,
Australian National University,
Canberra.

Department of Pure Mathematics,
University of Sydney,
Sydney.

Abstract. Collection processes have been the basis of group investigations by many people, some using hand calculation, some machine calculation. We describe a collection process which is specially efficient in the context of nilpotent quotient algorithm programs. The principles underlying our collection process are applicable in general.

1. Introduction. A collection process is a method for converting a word (in a group) into a normal form. P. Hall [13] introduced the notion of "the collecting process" and a definition of a collection process in the context of nilpotent quotient algorithms is given by M.F. Newman [20]. Since the principles underlying our collection algorithm are generally applicable, especially for machine implementation, we mention all machine implementations of collection processes of which we are aware. It is likely that machine implementations which do not apply our principles could be significantly improved.

The first machine implementation was by H. Felsch [9] with a program for collecting words of the form $(ab)^n$. Then J.M. Campbell and W.J. Lamberth [2] wrote a general purpose program for collecting words in free nilpotent groups of finite class and rank.

More recently K. Dařek [6], [7], [8] and E. Czyzo [3] have determined some exponents in various Hall formulas by combinatorial means (ultimately based on collection) and by direct collection. As far as direct collection is concerned they use programs described by E. Czyzo [4], [5].

From another point of view, H. Jürgensen [15] described a method for calculation with elements of a finite group given by generators and defining relations of special forms (based on work of J. Neubüser [19] and W. Lindenberg [16], [17]). He gave methods for multiplying and inverting such elements using what in fact is a collection process. V. Felsch [10] describes a new implementation of these methods in machine independent FORTRAN.

T.W. Sag and J.W. Wamsley [21] have described a method for computing the Schur multiplier of a finite supersoluble group. Their method involves multiplication of group elements which they perform by a collection process.

I.D. Macdonald [18] and J.W. Wamsley [22] have developed algorithms which, given a presentation of a group G and a positive integer c, construct the largest nilpotent of class c quotient group of G. Computer programs implementing these algorithms for specific p-quotients have been developed by Macdonald and Wamsley, and later by Alford, Havas and Newman [1], W. Felsch [11] and Havas-Newman (see Newman [20]). Collection plays a fundamental role in each of these programs.

2. Two methods of collection. The collecting process of P. Hall is amplified in M. Hall [12, §11.1]. In this kind of collecting process the basic step is the selection of the leftmost occurrence of the least uncollected letter and the collection of this letter relative to the letter on its left. We call this collection to the left.

Neubüser was the first to describe the effective collection process now described by V. Felsch. In this kind of collecting process the basic step is the selection of the rightmost uncollected subword and the collection of this subword. We call this collection from the right.

Each collecting process comprises the repeated performance of its basic step. In general neither collecting process terminates. To ensure termination practical collections are done modulo some term in a central series.

H. Felsch, Campbell and Lamberth, Czyzo, Macdonald and W. Felsch have all implemented variations of collection to the left. Neubüser, Lindenberg, Jürgensen, Sag and Wamsley, Wamsley, V. Felsch, Havas and Nicholson have all implemented variations of collection from the right.

3. Comparison. Our implementation of collection is strongly influenced by the work of Wamsley. He asserts that in practice collection from the right is better than collection to the left in the sense that it generally involves fewer steps and shorter intermediate words. We support this claim.

In the context of collection relative to a prime-power-commutator presentation on n generators with prime p, Newman has shown that there is a useful bound on the length of any word occurring at any stage, using collection from the right. Under the above conditions, if the original word to be collected has length l, then in the process of collection from the right no word of length greater than $l - 2 + (p-1)(2n-1)$ ever occurs. In general this bound is dramatically exceeded in collections to the left.

The claim that collection from the right is

better than collection to the left is illustrated by two different nilpotent quotient algorithm programs. (In significant applications more than 90% of computer time is spent doing collections.) The Canberra version, incorporating the collection algorithm described in this paper, took 23.67 seconds on a Univac 1108 to calculate the complete nilpotent quotient of a group of order 2^{21} and class 9, given a defining presentation including long eighth power relations. Another nilpotent quotient algorithm program, using collection to the left, given 10 minutes on a comparable machine, ran out of time while collecting the eighth powers at class 3. It did not complete the calculation of class 3. (Note that combinatorial collection as described in §4 has no beneficial influence for 2-groups, so this is a valid comparison.)

Consider the following presentation for the class 3 quotient of the above group (for abbreviation we represent a_i by i and omit trivial squares and commutators)

$$G = \langle 1, 2, 3, 4, 5, 6 \mid 2^2 = 4,\ 3^2 = 5,\ 4^2 = 5;\ [2, 1] = 3,\ [3, 1] = 5,\ [3, 2] = 6,\ [4, 1] = 56 \rangle .$$

To give yourself a feeling for collection, collect $(12)^4$ to the left and from the right, with respect to G. Using the most basic forms of collection and counting each move or square elimination as a step, you end up with 182 steps using collection to the left and 16 steps using collection from the right. The longest word arising in the course of collection to the left is of length 27, while the longest word arising in collection from the right is of length 10. With increasing complexity of word to be collected and increasing complexity of the presentation the difference is magnified.

With collection to the left improved by eliminating squares when they arise, there are still 32 steps involved. Collecting $(12)^8$ to the left this improved way takes 251 steps, with longest word of length 35, as against 32 steps for collection from the right, with longest word of length 18.

From the above example it follows that a binary decomposition of the exponent would be worthwhile for collection of $(12)^8$. There is scope for investigation of how a word to be collected should be preprocessed.

4. Improvements.

Lindenberg, Jürgensen and Wamsley observed that a major part of collection time was spent collecting expressions of the form yx^m and $y^n x^m$. Lindenberg and Jürgensen used presentations which include relations giving the values of $[y, x^m]$ for all necessary x, y and m, or alternatively computed these systematically prior to performing other collections. Wamsley went further in the computation of the 2 generator restricted Burnside group of exponent 5 [14] when he computed and stored the values of all commutators $[y^n, x^m]$.

The disadvantage of this kind of approach is that storage requirements are dramatically increased. For a p-quotient calculation storing additional tables multiplies the memory requirements by $(p\text{-}1)$ or $(p\text{-}1)^2$ depending on whether we store commutators of the form $[y, x^m]$ only or all commutators of the form $[y^n, x^m]$.

In our implementation we augment collection from the right by what we call combinatorial collection, to do collections of the above forms as quickly as possible in the absence of additional tables.

Consider yx^m where $y > x$, x has weight u, y has weight v. Then (by a combinatorial argument):

$$(1)\quad yx^m \equiv x^m y [y, x]^{\binom{m}{1}} [y, x, x]^{\binom{m}{2}} \cdots \cdots [y, \underbrace{x, \ldots, x}_{m}]^{\binom{m}{m}} \text{ modulo } \gamma_{2u+2v} .$$

Likewise:

$$(2)\quad y^n x^m \equiv x^m y^n [y, x]^{n\binom{m}{1}} [y, x, x]^{n\binom{m}{2}} \cdots \cdots [y, \underbrace{x, \ldots, x}_{m}]^{n\binom{m}{m}} \text{ modulo } \gamma_{u+2v} .$$

We use these formulas whenever applicable in our collection process. (It is possible to construct other formulas like (1) and (2) for calculating yx^m and $y^n x^m$ modulo different subgroups. We have not investigated these possibilities. Note that formula (1) is in fact correct modulo γ_{3u+2v}, but we only use it modulo γ_{2u+2v}. The reason for this is that if $[y, x] = \prod a_i^{\alpha_i}$ (in collected form) then we calculate $[y, x]^m$ by calculating $\prod a_i^{m\alpha_i}$ and $[y, x]^m \equiv \prod a_i^{m\alpha_i}$ modulo γ_{2u+2v}, but not necessarily modulo higher terms.)

To illustrate the significance of combinatorial collection consider the following timing tests, run with an early version of the Canberra nilpotent quotient algorithm program with collection from the right without combinatorial collection, then with the full collection process including combinatorial collection. The only difference between the two programs was the collection routine.

Let

$$G = \langle a, b \mid a^5 = b^5 = (ab)^5 = (a^2b)^5 = (ab^2)^5 = 1 \rangle .$$

The computation of the maximal 5-quotient of class 7 (order 5^{26}) took 17.14 seconds without combinatorial collection and took 2.57 seconds with combinatorial collection. For a maximal class 7-group the computation to the end of class 10 (order 7^{11}) took 12.53 seconds without as against 2.67 seconds with combinatorial collection.

5. Algorithm description.

5.1. Introduction. This algorithm description is made in one of the contexts in which the algorithm has been implemented, that is in the context of the Canberra nilpotent quotient algorithm program. (For other contexts some modifications are appropriate, but the same principles apply. Note, for example,

that this collection algorithm handles positive words only, because negative exponents are processed elsewhere in the nilpotent quotient algorithm program.) The collection described here is collection relative to a prime-power-commutator presentation:

$$\Big\langle a_1, \ldots, a_n \mid a_i^p = \prod_{k>i} a_k^{\alpha(i,k)} \quad , \quad 1 \le i \le n \; ;$$

$$[a_j, a_i] = \prod_{k>j} a_k^{\alpha(i,j,k)} \quad , \quad 1 \le i < j \le n \Big\rangle \; .$$

First it is necessary to describe the data structures. (This description is consistent with the data structures used in the Canberra implementation of the nilpotent quotient algorithm.)

Each word to be collected and all commutators and powers of the prime-power-commutator presentation are stored as strings of generator-exponent pairs. A string has a base address (*string*) and length (l) . Strings are processed from the right, and l is decremented as the string is processed, so that the l-th generator-exponent pair is the pair actually processed.

The collected result is computed as a vector of exponents (*exponent*), one exponent for each generator of the presentation. In the course of collection from the right this vector represents a number of normal word segments (not a collected part as is obtained at the left in collection to the left).

Thus all words are represented effectively with generator-exponent pairs. However the algorithmic treatment of the words is as free group words, that is $\prod_i a_i^{\alpha_i}$ is treated as $\prod_i \underbrace{a_i \ldots a_i}_{\alpha_i}$.

In collection from the right we find the rightmost uncollected subword (in this context either $a_j a_i$ with $j > i$ or a_i^p) and collect it. To collect $a_j a_i$ we move a_j past a_i to get $a_i a_j [a_j, a_i]$. This can lead to a new uncollected subword to the right of the letter we are moving (a_j) . The new uncollected subword needs to be processed before we finish moving a_j to its final position.

Thus this collection can be considered a recursive process. We achieve recursion by using a stack (*stack*) on which we save "the state of the collector" before starting to move a new letter to the right of one which is already in motion.

This algorithm is economical with storage because in addition to the presentation itself and the string to be collected all that is required is a small number of working variables and the stack. Copies are not made of any of the strings and the depth of the stack is limited (by the class of the presentation, in collection relative to a power-commutator presentation).

5.2. Variables used. In this collection algorithm there are eight main working variables, in four pairs: *string*, l ; *ug*, *ue* ; *cg*, *ce* ; m, r . In the course of collection *string*, l represent the string being processed. *ug*, *ue* represent the uncollected generator being moved and the number of occurrences (i.e. exponent) of *ug* which remain to be moved from the l-th generator-exponent pair of *string*. *cg*, *ce* represent the position, relative to the exponent vector, of those occurrences of *ug* actually being moved (in the sense that those occurrences of *ug* precede *ce* occurrences of generator *cg*). The position represented is the start of the trailing normal word segment. m is the actual number of occurrences (multiplicity) of *ug* being moved and r is a divisor used in computing combinatorial coefficients for evaluation of m from formulas.

The stack is "eight wide", in the sense that each of these eight variables is stored to represent the state of the collector. The depth of the stack is represented by a stack pointer (*sp*) . The variables stored on the stack have the same significance as described above, except that *cg*, *ce*, m , and r refer to the unprocessed part of *string*, that is everything up to and including the l-th generator-exponent pair, bar those occurrences of *ug* already being moved.

The significance of the other variables is as follows: p is the prime of the prime-power-commutator presentation; *class* is the class of the presentation; n is the number of generators in the presentation; *splim* is the stack level at which the change over from ordinary to combinatorial collection is made; *noofugmoved* indicates how many occurrences of *ug* (out of the *ue* available) we are actually moving; *noofugmoved* = 0 indicates that we have not yet determined how many we are moving, *noofugmoved* = 1 indicates that we are moving one occurrence of *ug* , while *noofugmoved* = -1 indicates that we are moving all *ue* occurrences of *ug* . (It is at the changeover level, *sp* = *splim* , that *noofugmoved* matters. Below that level we are doing ordinary collection so that we are moving only one *ug* , above that level we are doing full combinatorial collection so that we are moving all *ue* occurrences of *ug* .)

5.3. Break up of the algorithm. The algorithm can be broken up into three parts: ordinary collection; evaluation of p-th powers; and combinatorial collection. To follow the algorithm it is probably easiest to work level by level.

The lowest level, ordinary collection (from the right) without evaluation of p-th powers, involves steps 1, 2, 3, 5, 11, 12, 13, 14, 15 and 16 only. The variables m, r, *splim* and *noofugmoved* can basically be ignored. For this case $m = r = 1$ always, let $p = \infty$, let *sp* = *splim* in steps 13 and 14 , and let *noofugmoved* = 1 .

For the next level, ordinary collection with evaluation of p-th powers, reset p and allow steps 17 and 18 . (Note that in practice evaluation of some p-th powers is delayed to the end, so that this is not literally collection from the right.)

For combinatorial collection first, for simplicity, ensure that *noofugmoved* is not -1 (by setting it to 1 in step (6)) and allow all steps of the algorithm subject to this condition. Finally remove this restriction and you have the full algorithm. Observe that once we commence combinatorial collection we continue combinatorially till we return to the changeover level (*sp* = *splim*) of the stack.

5.4. The algorithm.

(1) (Initiallize collector) *exponent* (*) ← already collected part; *string*, *l* ← word to premultiply the already collected part; *sp* ← 0 ; *cg* ← 0 ; *ce* ← 1 (the whole of *string* is in front of the exponent vector which corresponds to one normal word segment); *m* ← 1 ; *r* ← 1 .

(1a) *ug*, *ue* ← *l*-th generator-exponent pair of *string*.

(2) (Save state of collector) *sp* ← *sp* + 1 ; *stack* (*sp*) ← *string*, *l*, *ug*, *ue*, *cg*, *ce*, *m*, *r* ; *noofugmoved* ← 0 (note that we have not yet determined how many of the *ue* occurrences of *ug* we will move).

(3) (Process next letter in the trailing normal word segment) *ce* ← *ce* - 1 ;

(3a) if *ce* > 0 (at least one occurrence of *cg* is to the right of *ug*) go to (4); (3b) (no occurrence of *cg* is to the right of *ug* so proceed to the next generator-exponent pair of the trailing normal word segment) *cg* ← *cg* + 1 ; *ce* ← *exponent* (*cg*) ; if *cg* < *ug* (*ug* and *cg* are out of order provided *cg* actually occurs, i.e. provided *ce* > 0) go to (3a); else (*ug* = *cg* , i.e. *ug* has reached its correct position relative to the exponent vector) *splim* ← *sp* (note that this stack level is the change over level from ordinary collection to combinatorial collection); (since *ug* has reached its correct position, *ug* is to be inserted into the exponent vector) go to (11).

(4) (We are going to move *ug* past *cg* . Check if we can switch to combinatorial collection.) If 2 × (weight(*ug*)+weight(*cg*)) > *class* (combinatorial collection is applicable) go to (6).

(5) (Use ordinary collection) If [*ug*, *cg*] is trivial go to (3b) (continue moving *ug*); (at this stage we have a new string [*ug*, *cg*] to be inserted after *cg*.*ug* . The new string is not in its correct position. By simple recursion we process this new string and return to the current string when we have finished with the new string) *string*, *l* ← [*ug*, *cg*]; to to (1a).

(6) (Commence combinatorial collection) *splim* ← *sp* (note that the current level is the changeover level).

(7) (Check if *ug* can be put into its correct position without creating any more commutators) If weight(*ug*) + weight(*cg*) > *class* (*ug* commutes with everything following it, so we can put the appropriate number of occurrences of *ug* in their correct position) go to (11).

(8) (Use combinatorial collection) If [*ug*, *cg*] is trivial go to (10b); (as in step (5) we have a new string [*ug*, *cg*] to insert not in its correct position); if *noofugmoved* = 0 (we have not yet determined how many occurrences of *ug* we can move together, so we do so now) then {if 2 × weight(*ug*) + weight(*cg*) > *class* (full combinatorial collection is applicable) *noofugmoved* ← -1 (note that we will move all *ue* occurrences of *ug* together), *m* ← *ue* (set multiplicity for *ug*); else *noofugmoved* ← 1 (we must move *ug*'s past *cg* one at a time and *m* remains set at 1)} ; {collect $ug^m\, cg^{ce}$ by term-wise formula evaluation. During the evaluation of terms in the formula *m* occurrences of *ug* are moved past all *ce* occurrences of *cg* in one step, but other terms are left behind} *string*, *l* ← [*ug*, *cg*] ; *ug*, *ue* ← *l*-th generator-exponent pair of *string* ; *m* ← (*m*×*ce*)/*r* (compute multiplicity of *string* in the evaluation of the formula); *r* ← *r* + 1 (compute divisor for multiplicity of the next term, if any, in the formula evaluation).

(9) (Save state of collector) *sp* ← *sp* + 1 ; *stack*(*sp*) ← *string*, *l*, *ug*, *ue*, *cg*, *ce*, *m*, *r* ; *m* ← *m* × *ue* (compute multiplicity for *ug*).

(10) (Find the next letter to be moved past This step is analogous with step (3), q.v. for comments.) *ce* ← *ce* - 1 ; (10a) if *ce* > 0 go to (7); (10b) *cg* ← *cg* + 1 ; *ce* ← *exponent*(*cg*) ; *r* ← 1 (we are on to a new generator-exponent pair in the trailing normal word segment, so we reset the divisor for a new formula evaluation); if *cg* < *ug* go to (10a) else go to (11).

(11) If *noofugmoved* = 0 (we have not yet determined how many occurrences of *ug* to move, so *ug* has commuted with everything between its initial position and its correct position) *noofugmoved* ← -1 , *m* ← *ue* (we can move all *ue* occurrences of *ug*); (*m* occurrences of *ug* have reached their correct position, so we can insert them in the exponent vector) *exponent*(*ug*) ← *exponent*(*ug*) + *m* ; if *exponent*(*ug*) ≥ *p* and 2 × weight(*ug*) < *class* go to (17) {ug^p occurs and must be evaluated now because it could cause more collections to the right of the current position. Observe that any exponent that is allowed to build up is at worst involved in combinatorial collection, never normal collection, so the exponent build up does not slow the process.}

(12) (We have finished with *m* occurrences of *ug* so we are ready to consider the previous letter, if any, in the current string) *cg*, *ce* ← *stack*(*sp*) (recall the position relative to the trailing normal word segment of any unprocessed part of *string*).

(13) (Find the previous letter on *string* , if any. This could be another occurrence of *ug* from the current generator-exponent pair.) *ue* ← *ue* - 1 ; if *ue* = 0 (there are no more occurrences of *ug* from this generator-exponent pair) go to (14) (proceed to previous generator-exponent pair of *string* , if any); if *sp* > *splim* (we were doing combinatorial collection, but not at the changeover level, so we moved all occurrences of *ug* from this generator-exponent pair together) go to (14); if *sp* = *splim* and *noofugmoved* = -1 (we were at the changeover level and we did move all occurrences of *ug* from this generator-exponent pair together) go to (14); (we have moved only one occurrence of *ug* and at least one remains) if *sp* < *splim* (we have just finished collecting a *p*-th power) *noofugmoved* ← 1 (note that we are moving one occurrence of *ug*); go to (15).

(14) (Make the preceding generator-exponent pair of *string* , if any, current) *l* ← *l* - 1 ; if *l* = 0 (we have reached the beginning of *string*) go to (16) (return to the previous collection); *ug*, *ue* ← *l*-th generator-exponent pair of *string* ; *stack*(*sp*) ← *l*, *ug* (update the state of the collector, all other stack entries, except *ue* , are correct, and *ue* is updated in step (15); if *sp* ≤ *splim* (we are in ordinary collection) *noofugmoved* ← 0 (note that we have not determined how many occurrences of *ug* we can move together).

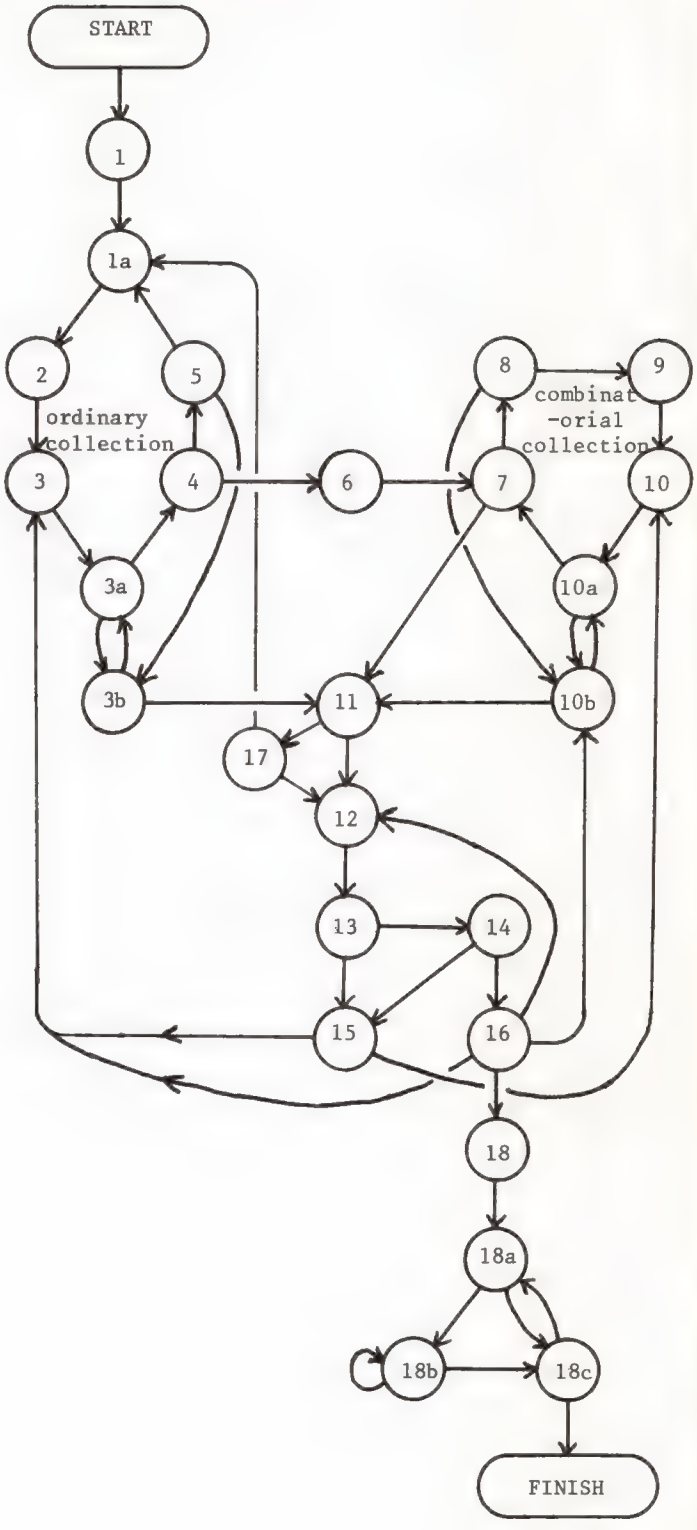
(15) (Process the next occurrence of ug from the current generator-exponent pair) $stack(sp) \leftarrow ue$ (update the state of the collector, all other stack entries are correct); $m, r \leftarrow stack(sp)$ (recover the multiplicity of the current string); if $sp \leq splim$ (we are using ordinary collection or at the changeover level) go to (3) (return to ordinary collection); else (we are using combinatorial collection) $m \leftarrow m \times ue$ (calculate the full multiplicity of ug); go to (10) (return to combinatorial collection).

(16) (We have finished processing the current string remembered in $stack(sp)$, so go back to the previous string, if any) $sp \leftarrow sp - 1$; if $sp = 0$ (we have virtually finished the entire collection, it remains to tidy up p-th powers in the exponent vector) go to (18); $ug, ue, string, l, m, r \leftarrow stack(sp)$ (restore previous collection in progress); if $ce = 0$ (last string was generated by a p-th power) go to (12) (return to where the p-th power came from); if $sp < splim$ (we are back to ordinary collection moving a letter that has already been moved, causing the introduction of an additional commutator) $noofugmoved \leftarrow 1$ (note that we have moved one occurrence of ug), go to (3) (return to normal collection); (we are back to combinatorial collection) if $sp > splim$ or $noofugmoved = -1$ (we are back to full combinatorial collection) $m \leftarrow m \times ue$ (calculate total multiplicity of ug); go to (10b) (return to combinatorial collection).

(17) [Evaluate ug^p] $exponent(ug) \leftarrow exponent(ug) - p$ (remove the p occurrences of ug that we are about to collect from the exponent vector); if ug^p is trivial go to (12) (return to where the p-th power came from). If $noofugmoved = -1$ (we are moving all occurrences of ug from this generator-exponent pair) $stack(sp) \leftarrow ue \leftarrow 1$ (alter the stored ue so that when we return we know that all occurrences of ug have been moved); $m \leftarrow 1$; $r \leftarrow 1$ [reset parameters for collection of ug^p]; $cg \leftarrow ug$; $ce \leftarrow 0$ [note position of ug^p relative to the exponents, $ce = 0$ indicates that this is a p-th power collection, and is used in step (16)]; $string, l \leftarrow ug^p$; go to (1a) (return to collection).

(18) (Tidy up the exponent vector by removing any exponents greater than p) $cg \leftarrow$ first generator with 2 × weight $\geq class$ (for each generator with 2 × weight $\geq class$ we allow the exponents to exceed p because the p-th power commutes with all subsequent generators so can be directly incorporated into the exponent array); (18a) $ce \leftarrow exponent(cg)$; if $ce < p$ go to (18c); $exponent(cg) \leftarrow exponent(cg) \bmod p$; if cg^p is trivial go to (18c); $m \leftarrow$ integer part of (ce/p) (calculate number of p-th powers to insert into the exponent array); $string, l \leftarrow cg^p$; (18b) ug, $ue \leftarrow l$-th generator-exponent pair of $string$; $exponent(ug) \leftarrow exponent(ug) + ue \times m$ (incorporate appropriate number of occurrences of ug into exponent vector); $l \leftarrow l - 1$; if $l > 0$ go to (18b) (process previous generator-exponent pair of $string$); (18c) $cg \leftarrow cg + 1$; if $cg \leq n$ go to (18a) (process next generator-exponent pair of the exponent vector); (we now have a vector of exponents corresponding to the collected result).

5.5. Flow diagram.



REFERENCES

[1] William A. Alford, George Havas and M.F. Newman, "Groups of exponent four", *Notices Amer. Math. Soc.* 22 (1975), A-301.

[2] J.M. Campbell and W.J. Lamberth, "Symbolic and numeric computation in group theory", *Proc. Third Austral. Computer Conf.* (Canberra, 1966), 293-296 (Australian Trade Publications, Chippendale, Australia, 1967). CR8#12,454.

[3] E. Czyżo, "On the determination of Hall polynomials", *Bull. Acad. Polon. Sci. Sér. Sci. Math. Astronom. Phys.* 23 (1975), 739-745.

[4] Emanuel Czyżo, "An attempt of mechanization of Hall collecting process", *Algorytmy* 9 (1972), 5-17. MR46#5419.

[5] E. Czyżo, "An automatization of the commutator calculus", *Algorytmy* 10 (1973), 23-34. MR48#1509.

[6] K. Dałek, "Computation of exponents in Hall formula", *Bull. Acad. Polon. Sci. Sér. Sci. Math. Astronom. Phys.* 19 (1971), 711-718. MR48#11327.

[7] K. Dałek, "On certain general forms of the exponents of simple commutators in Hall formula", *Bull. Acad. Polon. Sci. Sér. Sci. Math. Astronom. Phys.* 20 (1972), 973-980. MR48#11338.

[8] K. Dałek, "On the Hall formula in nilpotent and solvable groups", *Bull. Acad. Polon. Sci. Sér. Sci. Math. Astronom. Phys.* 23 (1975), 829-837.

[9] H. Felsch, "Die Behandlung zweier gruppentheoretischer Verfahren auf elektronischen Rechenmaschinen", Diplomarbeit, Kiel, 1960.

[10] Volkmar Felsch, "A machine independent implementation of a collection algorithm for the multiplication of group elements", *These Proceedings*.

[11] Waltraud Felsch, "Ein commutator collecting Algorithmus zur Bestimmung einer Zentralreihe einer endlichen p-Gruppe", Staatsexamensarbeit, Aachen, 1974.

[12] Marshall Hall, Jr., *The Theory of Groups* (The Macmillan Co., New York, 1959). MR21#1996.

[13] P. Hall, "A contribution to the theory of groups of prime-power order", *Proc. London Math. Soc.* (2) 36 (1934), 29-95.

[14] George Havas, G.E. Wall, and J.W. Wamsley, "The two generator restricted Burnside group of exponent 5 ", *Bull. Austral. Math. Soc.* 10 (1974), 459-470.

[15] H. Jürgensen, "Calculation with the elements of a finite group given by generators and defining relations", *Computational Problems in Abstract Algebra* (Proc. Conf. Oxford, 1967), 47-57 (Pergamon, Oxford, 1970). MR41#3600.

[16] Wolfgang Lindenberg, "Über eine Darstellung von Gruppenelementen in digitalen Rechenautomaten", *Numer. Math.* 4 (1962), 151-153. MR26#7174, CR4#4007.

[17] W. Lindenberg, "Die Struktur eines Übersetzungsprogramms zur Multiplikation von Gruppenelementen in digitalen Rechenautomaten", *Mitt. Rh.-W. Inst. Instr. Math. Bonn* 2 (1963), 1-38.

[18] I.D. Macdonald, "A computer application to finite p-groups", *J. Austral. Math. Soc.* 17 (1974), 102-112.

[19] J. Neubüser, "Bestimmung der Untergruppenverbände endlicher p-Gruppen auf einer programmgesteuerten elektronischen Dualmaschine", *Numer. Math.* 3 (1961), 271-278. MR24#B489.

[20] M.F. Newman, "Calculating presentations for certain kinds of quotient groups", *These Proceedings*.

[21] T.W. Sag and J.W. Wamsley, "On computing the minimal number of defining relations for finite groups", *Math. Comp.* 27 (1973), 361-368.

[22] J.W. Wamsley, "Computation in nilpotent groups (theory)", *Proc. Second.Internat. Conf. Theory of Groups* (Canberra, 1973), 691-700 (Lecture Notes in Mathematics, 372. Springer-Verlag, Berlin, Heidelberg, New York, 1974). MR50#7350.

Polynomial Real Root Isolation by Differentiation

George E. Collins[1]
Rüdiger Loos[2]

Abstract

A new algorithm is described, for the isolation of the real roots of a real polynomial. This algorithm utilizes the sequence of derivatives of the given polynomial, relying on Rolle's theorem and a tangent construction to decide whether an interval contains two roots or none. The algorithm is carefully compared, both analytically and empirically, with an algorithm based on Sturm's theorem, and is found to be significantly faster in general. It also requires less memory, and produces isolating intervals for the derivatives as a cost-free by-product.

1. Introduction

The real roots of a polynomial A are isolated by a set of disjoint intervals, each of which contains exactly one root of A, and which together contain all roots of A. Sturm's theorem, with interval bisection and exact arithmetic in the coefficient domain D, provides one infallible algorithm for real root isolation, which has been treated in detail by Heindel in [6] for the case D = Z, the ring of the natural integers. In the present paper we describe a new real root isolation algorithm which uses the sequence of derivatives of A instead of a Sturm sequence for A. The algorithm proceeds inductively from a set of isolating intervals for the derivative, A', to a set of isolating intervals for A.

In general, the coefficients of the sequence of derivatives of A are much smaller than the coefficients of a Sturm sequence for A. As a result, the new algorithm requires much less memory than the Sturm sequence algorithm, an effect which is further magnified by a device which avoids keeping more than two derivatives in memory at any time. Furthermore, the smaller coefficient size reduces the time required to compute and evaluate the sequence of polynomials. Accordingly, the new algorithm is generally much faster than the Sturm algorithm. This conclusion, with some minor exceptions, is supported by numerous empirical and analytical comparisons.

(1) Computer Sciences Department, University of Wisconsin, Madison, Wisconsin 53706, U.S.A.
Research supported in part by National Science Foundation grant DCR74-13278.
(2) Fachbereich Informatik, Universität Kaiserslautern, 6750 Kaiserslautern, West Germany.

Another advantage of the algorithm which will be described is that one obtains, without additional computing, isolating intervals for all derivatives of A. In [3] a quantifier elimination algorithm for the elementary theory of real closed fields is described, for which a real root isolation algorithm with this property is very useful. Also, a forthcoming paper [4] on high-precision polynomial real root calculation presents an application in which isolating intervals for the first and second derivatives are needed.

We will describe and analyze the new algorithm only for the case D = Z, but only trivial modifications are needed to apply it when D = Q, the field of rational numbers, or D = P, the field of real algebraic numbers. Mathematically, this remark applies to the Sturm sequence method also, but the computation of a Sturm sequence when D = P is extremely time consuming.

Conceptually, the algorithm is very simple. Basically, it applies Rolle's theorem, a simple tangent construction, and polynomial greatest common divisor calculations for the detection of multiple roots.

2. The Algorithm

The derivative sequence algorithm proceeds inductively from isolating intervals for the roots of A' (the derivative of A) to isolating intervals for the roots of A. For reasons to appear, it is required that the isolating intervals for A' contain no roots of A" other than roots common to A' and A". In order to proceed inductively, we then produce isolating intervals for A which contain no roots of A' other than common roots. Also, we associate with each isolating interval the multiplicity of the root which it contains. The multiplicities of the roots of A' are needed to make certain inferences about the graph of A. By induction one then obtains the multiplicities of the roots of A, which are important in many applications; the application to quantifier elimination is again a case in point. It should be noted that A' may have multiple roots even if the roots of A are all simple. As an important aspect of computational efficiency, the endpoints of the isolating intervals are required to be binary rational numbers, that is, rational numbers whose denominators are non-negative powers of 2. Also, we use intervals (a,b] which are left-open and right closed. Thereby, if $a < c < b$ then (a,b] is the disjoint union of the two

subintervals (a,c] and (c,b]. This is important in case A or one of its derivatives happens to have a binary rational root. We also find it convenient to keep the isolating intervals ordered, so that (a,b] precedes (c,d] in case $b \le c$. (Note that isolating intervals may have common endpoints.) According to these remarks, we say that a strong isolation list for A is a list $L = (I_1,m_1,\ldots,I_r,m_r)$ such that

(a) $r \ge 0$ is the number of distinct real roots of A.
(b) Each I_j is an interval $(a_j,b_j]$, where a_j and b_j are binary rational numbers.
(c) Each I_j contains a unique root, α_j, of A.
(d) m_j is the multiplicity of α_j as a root of A.
(e) $a_1 < b_1 \le a_2 < b_2 \le \ldots \le a_r < b_r$.
(f) A' has no roots in I_j except possibly α_j.

A root bound for the polynomial A is a real number γ such that if α is any root of A (real or complex) then $|\alpha| < \gamma$. A real root bound for A is a real number γ such that $|\alpha| < \gamma$ whenever α is a real root of A. In our algorithm we will need a real root bound for A and, because of the induction, a real root bound for each derivative of A. If $A(x) = \Sigma_{i=0}^{n} a_i x^i$ and $\deg(A) = n > 0$, then one root bound for A([8], Section 4.6.2, Exercise 20) is

$$\gamma = 2 \max_{1\le i\le n} |a_{n-i}/a_n|^{1/i}. \qquad (1)$$

(1) defines γ as a function f(A). Since $A'(x) = \Sigma_{i=0}^{n-1} i a_i x^{i-1}$ and $|(n-i)a_{n-i}/na_n| \le |a_{n-i}/a_n|$, we see that $f(A') \le f(A)$. Hence γ is a real root bound for each derivative for A. However, we have found that the total amount of computation in our algorithm is significantly reduced if one computes $f(A^{(k)})$ for each derivative, $A^{(k)}$, of A since one frequently has $f(A^{(k)}) < f(A)$ and this reduces the number of points at which $A^{(k)}$ will be evaluated.

Of course, we do not compute $|a_{n-i}/a_n|^{1/i}$ exactly. Instead we compute the least integer e_i such that $2^{e_i} \ge |a_{n-i}/a_n|^{1/i}$, equivalently $2^{ie_i}|a_n| \ge |a_{n-i}|$. If $2^{h-1} \le |a_{n-i}| < 2^h$, $2^{k-1} \le |a_n| < 2^k$ and $\ell = [(h-k-1)/i]$, then $e_i = \ell + 1$ or $e_i = \ell + 2$. Also, $e_i = \ell + 1$ unless $k - h \equiv 2 \pmod i$. Heindel, [6], computed a somewhat cruder approximation, γ^*, to γ, satisfying $\gamma^* \le 8\gamma$.

The 1-norm of the polynomial A is defined to be

$$|A|_1 = \Sigma_{i=0}^{n} |a_i|. \qquad (2)$$

If the coefficients a_i are integers then $|a_{n-i}/a_n|^{1/i} \le |a_{n-i}|^{1/i} \le |a_{n-i}|$, so $\gamma \le 2|A|_1$ and our approximation $\bar{\gamma}$ to γ satisfies

$$\bar{\gamma} \le 4|A|_1. \qquad (3)$$

This inequality will be used later in our analysis of the algorithm.

Our root isolation algorithm uses a subalgorithm, which we call S, for the induction step. The inputs to S are a polynomial A with $\deg(A) \ge 1$ and $\mathrm{ldcf}(A) > 0$ (ldcf(A) denotes the leading coefficient of A), its derivative, A', a binary rational real root bound, γ, for A, and a strong isolation list, L', for A'. The output of S is a strong isolation list, L, for A.

Let $L' = (I_1,m_1,\ldots,I_r,m_r)$, with $I_j = (a_j,b_j]$. The isolating intervals $I_1,\ldots,I_r$ for A', together with the root bound $\gamma = a_{r+1}$ for A and its negative $-\gamma = b_0$, define r + 1 complementary intervals $J_0,\ldots,J_r$, with $J_i = (b_i,a_{i+1}]$, as illustrated in Figure 1 for the case r = 3. We are assuming, without loss of generality, that $-\gamma \le a_1$ and $b_r \le \gamma$. We may have $b_i = a_{i+1}$, in which case J_i is the empty interval.

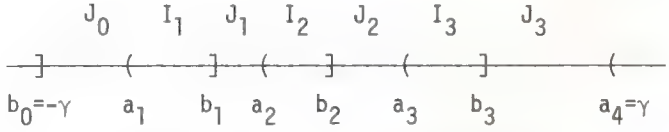


Figure 1. Isolating Intervals and Their Complements

As a special case we may have r = 0. Then A' has no real roots so deg(A') is even, deg(A) is odd, and hence A has at least one real root. If A had two real roots, then A' would have a root in between, contrary to assumption, so A has exactly one real root. Moreover, this root must be simple for otherwise it would also be a root of A'. Hence in this case $((-\gamma,\gamma],1)$ is a strong isolation list for A.

In general, each complementary interval $J_i = (b_i,a_{i+1}]$ contains at most one root of A and any such root is simple by a similar argument. Let

$$t_i^* = \mathrm{sign}(A(b_i+)) \qquad (4)$$

denote the sign of $A(b_i+\epsilon)$ for all sufficiently small $\epsilon > 0$, and let

$$s_i = \mathrm{sign}(A(a_i)). \qquad (5)$$

Then A has a root in $J_i = (b_i,a_{i+1}]$ just in case

$$t_i^* \cdot s_{i+1} \le 0. \qquad (6)$$

If this condition is satisfied, then J_i and the multiplicity 1 may be entered into a strong isolation list for A.

We define also

$$t_i' = \mathrm{sign}(A'(b_i+)). \qquad (7)$$

Since $\mathrm{ldcf}(A') > 0$, $\deg(A') = n-1$, and $\gamma = -b_0$ is a root bound for A',

$$t'_0 = (-1)^{n-1}. \tag{8}$$

Inductively,

$$t'_i = (-1)^{m_i} t'_{i-1} \tag{9}$$

for $i \geq 1$. Define also

$$t_i = \text{sign}(A(b_i)). \tag{10}$$

We then have

$$t^*_i = \left\{ \begin{matrix} t_i, \text{ if } t_i \neq 0 \\ t'_i, \text{ if } t_i = 0 \end{matrix} \right\}. \tag{11}$$

The interval $(b_{i-1},a_{i+1}] = J_{i-1} \cup I_i \cup J_i$ contains exactly one real root of A'; hence by Rolle's theorem it contains at most two roots of A. Therefore if J_{i-1} and J_i both contain roots of A then there are no roots of A in I_i.

If J_{i-1} and J_i do not both contain roots, then we next decide whether α_i, the unique root of A' in I_i, is a root of A. If it is a root of A then its multiplicity must be $m_i + 1$ since its multiplicity as a root of A' is m_i. Moreover, by Rolle's theorem it must then be the only root of A in I_i, so I_i and $m_i + 1$ may then be placed in a strong isolation list for A.

Let

$$\bar{A}' = \text{gsfd}(A') \tag{12}$$

be the greatest squarefree divisor of A'. The greatest squarefree divisor of a polynomial is defined by

$$\text{gsfd}(C) = C/\text{gcd}(C,C'). \tag{13}$$

gsfd(C) has the same roots as C, but its roots are all simple. By convention the greatest common divisor of two polynomials has a positive leading coefficient. Hence if ldcf(C) > 0 then ldcf(gsfd(C)) > 0. Also, gsfd(C) is always primitive. Thus ldcf($\bar{A}'$) > 0 and $\bar{A}'$ is primitive. Let

$$B = \text{gcd}(A,\bar{A}'). \tag{14}$$

B is primitive, ldcf(B) > 0, all roots of B are simple, and α_i is a root of A if and only if α_i is a root of B. Since there are no roots of B in $(b_{i-1},b_i]$ other than α_i, α_i is a root of A if and only if B has a root in $(b_{i-1},b_i]$, thus if and only if

$$w^*_{i-1} \cdot w^*_i < 0, \tag{15}$$

where

$$w^*_i = \text{sign}(B(b_i+)). \tag{16}$$

If deg(B) = k then

$$w^*_0 = (-1)^k, \tag{17}$$

and

$$w^*_i = \left\{ \begin{matrix} \text{sign } (B(b_i)), \text{ if } B(b_i) \neq 0 \\ -w^*_{i-1}, \text{ if } B(b_i) = 0 \end{matrix} \right\} \tag{18}$$

for $i \geq 1$.

We may now assume that α_i is not a root of A. If m_i is even then A is either monotone increasing in I_i as illustrated in Figure 2, or monotone decreasing as illustrated in Figure 3. In either case I_i contains exactly one root of A in case $s^*_i \cdot t_i \leq 0$, and no roots otherwise. Figures 2 to 9 refer to

$$s'^*_i = \text{sign}(A'(a_i+)). \tag{19}$$

We note that, since A' has no roots in $(b_{i-1},a_i]$,

$$s'^*_i = t'_{i-1}. \tag{20}$$

If either J_{i-1} or J_i contains a root of A, then I_i contains at most one root of A, which must be simple, so in this case also A has a root in I_i just in case $s^*_i \cdot t_i \leq 0$. If A has a unique root in I_i then a subinterval of I_i must be found which contains the root of A but not α_i. This is achieved by applying a subalgorithm, R, which will be described subsequently, to A, $\bar{A}'$, I_i, s^*_i and $\bar{s}'_i$, where

$$\bar{s}'_i = \text{sign}(\bar{A}'(a_i+)). \tag{21}$$

Figure 2. m_i even, $s^*_i \cdot s'^*_i > 0$ (no roots).

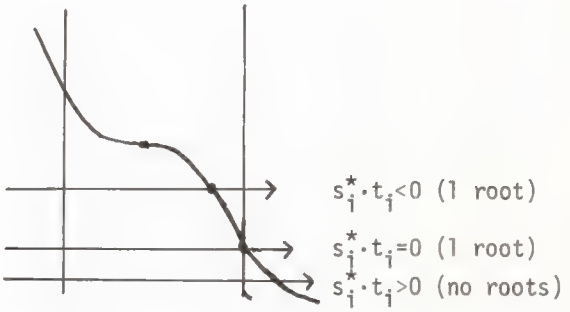


Figure 3. m_i even, $s^*_i \cdot s'^*_i < 0$.

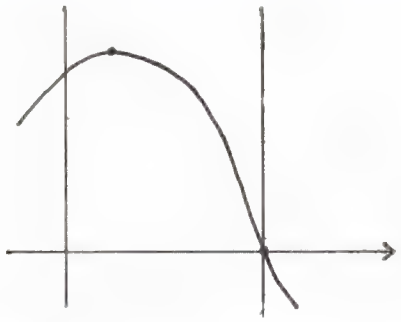


Figure 4. m_i odd,
$s_i^* \cdot s_i'^* > 0$, $t_i = 0$ (1 root).

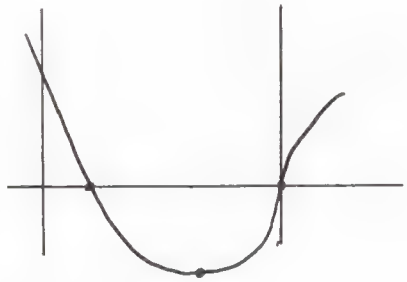


Figure 5. m_i odd,
$s_i^* \cdot s_i'^* < 0$, $t_i = 0$ (2 roots).

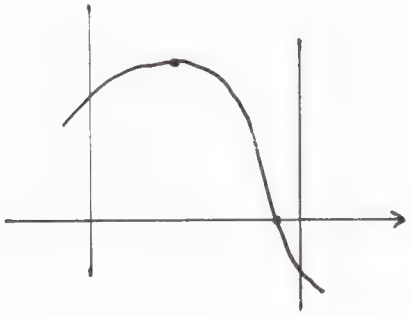


Figure 6. m_i odd,
$s_i^* \cdot s_i'^* > 0$, $s_i^* \cdot t_i < 0$ (1 root).

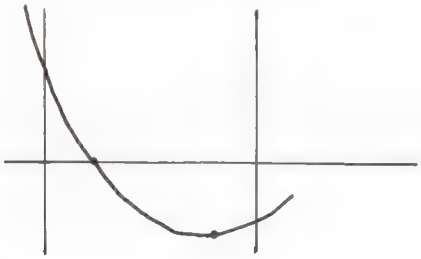


Figure 7. m_i odd,
$s_i^* \cdot s_i'^* < 0$, $s^* \cdot t_i < 0$ (1 root).

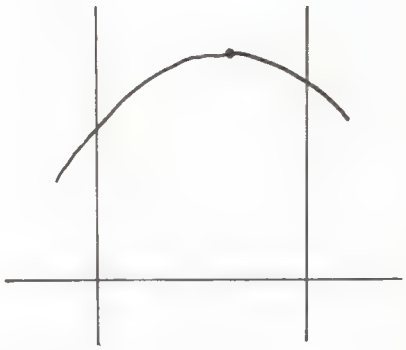


Figure 8. m_i odd,
$s_i^* \cdot s_i'^* > 0$, $s_i^* \cdot t_i > 0$ (no roots).

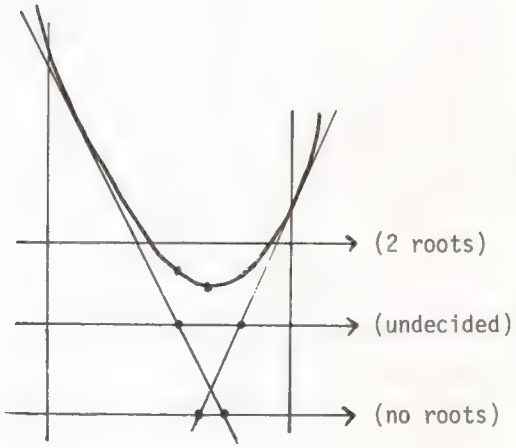


Figure 9. m_i odd,
$s_i^* \cdot s_i'^* < 0$, $s_i^* \cdot t_i > 0$.

If $m = \deg(\bar{A}')$, then

$$\bar{s}_1' = (-1)^m, \tag{22}$$

and

$$\bar{s}_{i+1}' = \begin{cases} \bar{s}_i', & \text{if } A(\alpha_i) \neq 0 \\ \bar{s}_i', & \text{if } A(\alpha_i) = 0 \end{cases} \tag{23}$$

for $i \geq 1$.

Now we may assume that $A(\alpha_i) \neq 0$ and m_i is odd. If $A(b_i) = 0$ then we have one of the two cases illustrated in Figures 4 and 5. We see that if $s_i^* \cdot s_i'^* > 0$ then b_i is the unique root of A in I_i and we can again apply algorithm R, while if $s_i^* \cdot s_i'^* < 0$ then A has two roots in I_i. In the latter case, we bisect $(a_i, b_i]$ into two subintervals, $(a_i, c]$, and $(c, b_i]$, and compute $s = \text{sign}(A(c))$. If $s_i^* \cdot s \leq 0$ then each subinterval contains one of the

roots, and algorithm R is applied to that subinterval containing α_i, as determined by sign (A'(c)). Otherwise, $(c,b_i]$ contains both roots of A, and the bisection process is repeated with $c,b_i]$ in place of $(a_i,b_i]$. Eventually, the two roots of A will be separated.

Now we may assume also that $A(b_i) \neq 0$. If $s_i^* \cdot t_i < 0$, then we have one of the two situations illustrated in Figures 6 and 7, in either of which A has exactly one root in I_i, and algorithm R is applied.

Suppose, finally, that $s_i^* \cdot t_i > 0$. If $s_i^* \cdot s_i'^* > 0$, then A has no roots in I_i, as illustrated by Figure 8. If, however, $s_i^* \cdot s_i'^* < 0$, then A may have either no roots or two roots in I_i, as shown in Figure 9, since we are assuming that the case $A(\alpha_i) = 0$ has already been excluded. In this case we compute A(c) for a bisection point c. If $s_i^* \cdot A(c) < 0$, then the subintervals $(a_i,c]$ and $(c,b_i]$ each contain one root of A, and algorithm R is applied to the subinterval containing α_i as determined by sign(A'(c)). If, however, $s_i^* \cdot A(c) > 0$, then we attempt to prove that there are no roots in the interval. We compute the intersection,

$$a_i' = a_i \leftarrow A(a_i)/A'(a_i), \tag{24}$$

of the axis and the tangent to A(x) at $x = a_i$, and the intersection

$$b_i' = b_i \leftarrow A(b_i)/A'(b_i), \tag{25}$$

of the axis and the tangent to A(x) at $x = b_i$. If $a_i' \geq b_i'$ then A has no roots in I_i because A" has no roots in I_i other than α_i, m_i is odd, and hence A' is monotone in I_i so that A does not cross its tangents within I_i. If, however, $a_i' < b_i$ (as in Figure 9) then that subinterval, $(a_i,c]$ or $(c,b_i]$, which does not contain α_i does not contain any roots of A and the other subinterval satisfies the same conditions as I_i. So I_i is replaced by the appropriate subinterval and the bisection and tangent construction process is repeated until two roots are discovered and separated or it is proved that none exist.

Algorithm R has five inputs: A, B, I, s and t. I is an interval (a,b]. A and B are polynomials having unique roots, α and β, in I, with $\alpha \neq \beta$. The multiplicities of α and β are both odd. s = sign(A(a+)) and t = sign(B(a+)). The output of algorithm R is a subinterval of (a,b], say (a',b'], containing α but not β. The algorithm uses repeated bisection. Let c be a bisection point, $\bar{s}$ = sign(A(c+)), $\bar{t}$ = sign(B(c+)), $u = s \cdot \bar{s}$ and $v = t \cdot \bar{t}$. If $u > 0$ then (c,b] replaces (a,b] and otherwise (a,c] replaces (a,b]. If u = v then the process is repeated and otherwise the new (a,b] is the output.

This completes the description of our algorithm in its main outline, but there are some other aspects requiring some discussion because the performance of the algorithm significantly depends on their proper treatment.

The most important of these aspects relates to the manner in which intervals are bisected. The most obvious and most natural way to bisect an interval is to use its midpoint, dividing it into two subintervals of equal length. Furthermore, this method preserves the property of having binary rational endpoints. However, it does not minimize the lengths of the numerators and denominators of the interval endpoints, and this is important in reducing the time required to evaluate the polynomial A and its derivatives at the endpoints.

Suppose, for example, that (0,1/16] and (1,2] are consecutive isolating intervals for A" and that the complementary interval (1/16,1] contains a root of A'. If there are two roots of A in (1/16,1], then we may obtain (1/16,17/32] and (17/32,1] as isolating intervals. If the first of these contains the root of A', then we may obtain (1/16,19/64], for example, as a subinterval containing the root of A but not the root of A'. It is better to choose 1/2 as the bisection point of (1/16,1]. If (1/16,1/2] and (1/2,1] each contains a root of A and the former contains the root of A' then we may bisect it with the point 1/4 and obtain (1/16,1/4] and (1/2,1] as our strong isolating intervals for A. The effect of using midpoints illustrated here becomes much more serious when one considers the effect through a long chain of derivatives.

How were the bisection points 1/2 and 1/4 chosen above? Consider an arbitrary interval (a,b] and let $2^h \leq b-a < 2^{h+1}$. Then there are either one or two points of the form $c \cdot 2^h$ in (a,b], c an integer, and we choose the least. As a special case this point may be equal to b. But then we must have $b-a = 2^h$ and we choose the midpoint $a + 2^{h-1}$ as bisection point.

Most of the computing time of Algorithm S is spent in computing A(a) or A'(a) for various binary rational numbers a. It is therefore important to have an efficient algorithm for computing A(a), given an integral polynomial A and a binary rational number a. We use an algorithm which makes use of the assumption that a is a binary rational. Exact integer arithmetic on arbitrarily large integers is performed by the SAC-1 integer arithmetic system, [1] and [2]. In this system integers are expressed in radix representation relative to a radix, or base, β. β is a system parameter which depends on the computer on which SAC-1 is running. We assume here that $\beta = 2^\zeta$ for some positive integer ζ. For the UNIVAC 1110 implementation of SAC-1 at the University of Wisconsin $\zeta = 33$. The assumption that $\beta = 2^\zeta$ contributes further to the efficiency of our algorithm for computing A(a).

If a = 0 then the computation of A(a) is trivial and is handled as a special case, so assume $a \neq 0$. By convention, rational numbers are kept in lowest terms with positive denominators, so

$a = e/f$ with $f = 2^k$, $k \geq 0$; if $k > 0$ then e is odd. We compute k from f with the aid of the assumption $\beta = 2^\zeta$. If $f = \Sigma_{i=0}^{m} f_i \beta^i$ with $f_m \neq 0$ then $k = m\zeta + h$ where $f_m = 2^h$; m and h are readily computed.

If $A(x) = \Sigma_{i=0}^{n} a_i x^i$, $a_n \neq 0$, then

$$f^n A(a) = \Sigma_{i=0}^{n} a_i e^i f^{n-i}. \tag{26}$$

The right hand side of (26) is computed by an obvious modification of Horner's method. Let

$$A_\ell = \Sigma_{i=\ell}^{n} a_i e^{i-\ell} f^{n-i}. \tag{27}$$

Then $A_n = a_n$,

$$A_\ell = eA_{\ell+1} + a_\ell f^{n-\ell} \tag{28}$$

for $0 \leq \ell < n$, and $A_0 = \Sigma_{i=0}^{n} a_i e^i f^{n-i}$. Since $f = 2^k$, $f^{n-\ell} = 2^{k(n-\ell)}$, and a_ℓ is multiplied by $2^{k(n-\ell)}$ in (28) by a special algorithm which is given a_ℓ and $k(n-\ell)$ as inputs. If $k(n-\ell) = q\zeta + r$, $0 \leq r < \zeta$, then a_ℓ is multiplied by the small integer 2^r and the result is multiplied by β^q merely by annexing q low-order zeros to the β-radix representation.

By (26), $A(a) = A_0/f^n$ and it remains only to reduce this fraction to lowest terms. Instead of applying a general integer greatest common divisor algorithm, this is accomplished by finding the largest integer t such that $t \leq kn$ and $2^t | A_0$; again one uses the assumption $\beta = 2^\zeta$.

Let $d = |A|_1$. It is easy to show that the computing time of the polynomial evaluation algorithm which we have just described is dominated (see [2] for a definition of dominance) by

$$n^2 L(e)^2 + n^2 L(e)L(f) + nL(d)L(e). \tag{29}$$

Here $L(a)$ denotes the length of the integer a, the number of bits in its binary representation, and we are assuming the use of a "classical" integer integer multiplication algorithm in the sense of [4]. If we apply the same modification of Horner's algorithm but without the assumptions $f = 2^k$ and $\beta = 2^\zeta$, we obtain in place of (29) the dominating function

$$n^2\{L(e)+L(f)\}^2 + n^2 L(d)\{L(e)+L(f)\}. \tag{30}$$

Let A be an integral polynomial with $\deg(A) = n > 0$ and $\mathrm{ldcf}(A) > 0$. Let $A^{(i)}$ be the ith derivative of A, and $\bar{A}^{(i)} = \mathrm{pp}(A^{(i)})$, the primitive part of $A^{(i)}$, defined by $\mathrm{pp}(A) = A/\mathrm{cont}(A)$ where $\mathrm{cont}(A)$ is the content of A. We compute the $\bar{A}^{(i)}$ using the relation $\bar{A}^{(i+1)} = \mathrm{pp}(\bar{A}^{(i)\prime})$ together with $\bar{A}^{(0)} = \mathrm{pp}(A)$. Since $\deg(\bar{A}^{(n)}) = 0$, the null list, L_0, is trivially a strong isolation list for $\bar{A}^{(n)}$. Given a strong isolation list, L_k, for $\bar{A}^{(k)}$, we compute a binary rational root bound, γ_{k-1}, for $\bar{A}^{(k-1)}$ and apply Algorithm S to $\bar{A}^{(k-1)}$, $\bar{A}^{(k-1)\prime}$, γ_{k-1} and L_k, obtaining a strong isolation list, L_{k-1}, for $\bar{A}^{(k-1)}$, for $k = n, n-1, \ldots, 1$. In order to avoid keeping the entire sequence $\bar{A}^{(0)}, \bar{A}^{(1)}, \ldots, \bar{A}^{(n)}$ in memory, we recompute the $\bar{A}^{(k)}$ by integration as needed. Let $b_k = \bar{A}^{(k)}(0)$ and $c_k = \mathrm{cont}(\bar{A}^{(k)\prime})$. Then $\bar{A}^{(k)\prime} = c_k \bar{A}^{(k+1)}$ and $\bar{A}^{(k)}$ is the integral of $\bar{A}^{(k)\prime}$ with b_k as constant of integration. Hence it suffices to store $b_0, b_1, \ldots, b_{n-2}$ and $c_0, c_1, \ldots, c_{n-2}$.

3. Computing Time Analysis

The ∞-norm of the polynomial $A(x) = \Sigma_{i=0}^{n} a_i x^i$ is defined by

$$|A|_\infty = \max_{0 \leq i \leq n} |a_i|. \tag{31}$$

The following theorem provides bounds for the norms of the $\bar{A}^{(k)}$.

Theorem 1. Let $\deg(A)$ $n > 0$. Then $|\bar{A}^{(k)}|_\infty \leq \binom{n}{k} |A|_\infty$ and $|\bar{A}^{(k)}|_1 \leq \binom{n}{k} |A|_1$.

Proof. Let $B_k(x) = A^{(k)}(x)/k!$. Then $B_k(x) = \Sigma_{i=k}^{n} \binom{i}{k} a_i x^{i-k}$, an integral polynomial. Hence $\bar{A}^{(k)} = \mathrm{pp}(B_k)$, $|\bar{A}^{(k)}|_\infty \leq |B_k|_\infty = \max_{k \leq i \leq n} \binom{i}{k} |a_i| \leq \max_{k \leq i \leq n} \binom{n}{k} |a_i| \leq \binom{n}{k} |A|_\infty$, and similarly $|\bar{A}^{(k)}|_1 \leq |B_k|_1 \leq \binom{n}{k} |A|_1$. □

We will assume throughout the following analysis that classical algorithms are used for integer arithmetic calculation. Also we set $n = \deg(A)$ and $d = |A|_\infty$. Using Theorem 1, we easily obtain:

Theorem 2. The time to compute all $\bar{A}^{(k)}$, and to recompute them by integration, is dominated by $n^4 + n^2 L(d)^2$.

Proof. By Theorem 1, $|\bar{A}^{(k)}|_\infty \leq \binom{n}{k} d \leq 2^n d$ and $|\bar{A}^{(k)\prime}|_\infty \leq 2^n nd$. Hence (since we make the modest assumption that $L(n) = 1$) $L(|\bar{A}^{(k)\prime}|_\infty) \leq n + L(d)$ and the time to compute $\mathrm{pp}(\bar{A}^{(k)\prime})$ is $\leq n\{n+L(d)\}^2 \sim n^3 + nL(d)^2$. Multiplying by n, the time to compute all primitive parts is $\leq n^4 + n^2 L(d)^2$. It is easy to see that all derivatives and integrals can be computed in time $\leq n^3 + nL(d)$. □

We note for comparison that the time to compute the Sturm sequence of A is (by [6]), $\leq n^5L(d)^2$, and $\leq n^4L(d)^2$ if the sequence is normal. Also, we could use $A^{(k)}/k!$ in place of $\bar{A}^{(k)}$, reducing the theoretical time of Theorem 2 from $n^4 + n^2L(d)^2$ to $n^3 + nL(d)$. But, as we will see in the following section, in some cases the $\bar{A}^{(k)}$ have much smaller coefficients.

Assuming again that the base $\beta = 2^\zeta$, it is easy to see how to compute $\lfloor \log_2(a) \rfloor$ in time $\leq L(a)$, from which the following theorem follows readily.

Theorem 3. Binary rational root bounds for all the $\bar{A}^{(k)}$ as defined by the discussion of Section 2 can be computed in time $\leq n^2L(d)$.

Application of Algorithm S to $\bar{A}^{(k)}$ and $\bar{A}^{(k)}$, results in several evaluations of $\bar{A}^{(k)}$ and $\bar{A}^{(k)}$, at binary rational points e/f. If γ_k is the computed root bound for $\bar{A}^{(k)}$ then $|e/f| \leq \gamma_k \leq \gamma_0 \leq |A|_1 \leq 2d$. Hence $L(e) \leq L(f) + L(d)$. Also, as we have seen, $L(|\bar{A}^{(k)}|), L(|\bar{A}^{(k)\prime}|) \leq n + L(d)$. Substituting $n + L(d)$ for $L(d)$ and then $L(f) + L(d)$ for $L(e)$ in (29), and simplifying, we arrive at

$$n^2L(d)^2 + n^2L(f)^2 \tag{32}$$

as a dominance bound for the time to evaluate $\bar{A}^{(k)}$ or its derivative at any one point. It remains to obtain a bound for f and a bound for the number of evaluations, in each case as a function of n and d. These bounds will be obtained by deriving a lower bound for the lengths of the isolating intervals produced by Algorithm S.

It is proved in [5] that the distance between any two distinct roots of a polynomial A with $\deg(A) = n$ and $|A|_1 = d$ is at least

$$\tfrac{1}{2}(e^{1/2}n^{3/2}d)^{-n}, \tag{33}$$

provided that A has only simple roots. If A has multiple roots, let $B = \mathrm{gsfd}(A)$, $m = \deg(B)$, $c = |B|_1$. By Mignotte's theorem, [9], $c \leq 2^m d$, so the distance between any two distinct roots of B, hence of A, is $\geq \frac{1}{2}(e^{1/2}m^{3/2}2^m d)^{-m} \geq \frac{1}{2}(e^{1/2}n^{3/2}2^n d)^{-n}$. Applying this result to $\bar{A}^{(k)}$, since $|\bar{A}^{(k)}|_1 < 2^n nd$ we obtain

$$\lambda = \tfrac{1}{2}(e^{1/2}n^{5/2}2^{2n}d)^{-n} \tag{34}$$

as a lower bound on the distance between any two roots of $\bar{A}^{(k)}$. Any interval shorter than λ contains at most one root of $\bar{A}^{(k)}$. If it contains exactly one root then this will be discovered by Algorithm S without further bisection. It remains to determine μ as a function of n and d such that if an interval contains no roots and is shorter than μ, then the tangent construction will succeed in proving that there are no roots.

Theorem 4. Let A(x) be an integral polynomial with $\deg(A) = n \geq 2$ and $|A|_1 = d$. Let $T(x) = A(a) + A'(a)\cdot(x-a)$ be the tangent to A(x) at $x = a$. Then $|A(x)-T(x)| \leq 2^{n+1}\bar{a}^{n-2}d\cdot|x-a|^2$ for $|x-a| \leq 1/2$, where $\bar{a} = \max(|a|,1)$.

Proof. By Taylor's theorem, $A(x) = \Sigma_{i=0}^{n}\{A^{(i)}(a)/i!\}(x-a)^i$. Hence $|A(x)-T(x)| \leq \Sigma_{i=2}^{n}\{|A^{(i)}(a)|/i!\}|x-a|^i$. But $|A^{(i)}(a)|/i! \leq |A^{(i)}|_1\cdot|a|^{n-i}/i! \leq \binom{n}{i}\cdot|A|_1\cdot|a|^{n-i} \leq 2^n d\cdot|a|^{n-i}$ so $\Sigma_{i=2}^{n}\{|A^{(i)}(a)|/i!\}|x-a|^2 \leq 2^n d\cdot|a|^{n-2}\cdot|x-a|^2\Sigma_{i=0}^{\infty}|x-a|^i \leq 2^n d\cdot|a|^{n-2}\cdot|x-a|^2\Sigma_{i=0}^{\infty}2^{-i} = 2^{n+1}|a|^{n-2}d\cdot|x-a|^2$ if $|a| \geq 1$ and $\Sigma_{i=2}^{n}\{|A^{(i)}(a)|/i!\}|x-a|^2 \leq 2^n d\Sigma_{i=2}^{n}|x-a|^i \leq 2^{n+1}d\cdot|x-a|^2$ if $|a| < 1$. □

Theorem 5. Let A and B be integral polynomials with $\deg(A) = m > 0$, $\deg(B) = n > 0$, $|A|_1 = d$ and $|B|_1 = e$. If $B(\beta) = 0$ and $A(\beta) \neq 0$ then $|A(\beta)| > \frac{1}{2}(d+1)^{-n}e^{-m}$.

Proof. Let R(x) be the resultant with respect to y of B(y) and x-A(y). By [10], Section 28, $R(x) = b_n^m\Pi_{i=1}^{n}(x-A(\beta_i))$, where $b_n = \mathrm{ldcf}(B)$ and $\beta_1,\ldots,\beta_n$ are the roots of B. Also, the degree of $x - A(y)$ in y is m and $|x-A(y)|_1 = d+1$. By [7] (Theorem 1), $|R|_\infty \leq (d+1)^n e^m$. Let $R(x) = x^k\Sigma_{i=0}^{n-k}r_i x^i$ with $r_0 \neq 0$ and set $R^*(x) = \Sigma_{i=0}^{n-k}r_i x^{n-k-i}$. Then $A(\beta)^{-1}$ is a root of R^* so by (1), $|A(\beta)^{-1}| < 2|R^*|_\infty = 2|R|_\infty \leq 2(d+1)^n e^m$ and $|A(\beta)| > \frac{1}{2}(d+1)^{-n}e^{-m}$. □

Theorem 6. Let A be an integral polynomial with $\deg(A) = m > 0$ and $|A|_1 = d$. If $A'(\beta) = 0$ and $A(\beta) \neq 0$ then $|A(\beta)| > (2m)^{-m}d^{-2m+1}$

Proof. Set $B = A'$ in Theorem 5, so that $n = m - 1$ and $e \leq md$. Then $|A(\beta)| > \frac{1}{2}(d+1)^{-m+1}(md)^{-m} \leq \frac{1}{2}(2d)^{-m+1}(md)^{-m} = (2m)^{-m}d^{-2m+1}$. □

Theorem 7. Let A be an integral polynomial with $\deg(A) = n \geq 2$ and $|A|_1 = d \geq 2$. Let $A'(\beta) = 0$ and $A(\beta) \neq 0$. Let T(x) be the tangent to A(x) at $x = a$. Let $|a| \leq d$ and $|\beta-a| \leq (nd)^{-2n}$ Then $A(\beta)$ and $T(\beta)$ have the same sign.

Proof. By Theorem 4, $|A(\beta)-T(\beta)| \leq 2^{n+1}d^{n-2}d(nd)^{-4n} \leq d^{2n}(nd)^{-4n} = n^{-4n}d^{-2n}$. By Theorem 6, $|A(\beta)| > (2n)^{-n}d^{-2n+1} \geq n^{-2n}d^{-2n} > |A(\beta)-T(\beta)|$. □

Since $|\bar{A}^{(k)}|_1 \le 2^n nd$, Theorem 7 shows that the tangent construction will always succeed when applied to $\bar{A}^{(k)}$ and an interval with length less than

$$\mu = 2^{-2n^2} n^{-4n} d^{-2n}, \tag{35}$$

since the cases $\deg(\bar{A}^{(k)}) = 1$ and $|\bar{A}^{(k)}|_1 = 1$ are trivial. If f is the denominator of any binary rational endpoint which is generated by Algorithm S, then it is not difficult to see that $L(f) \le \max\{L(\lambda^{-1}), L(\mu^{-1})\}$. Hence, by (34) and (35),

$$L(f) \le n^2 + nL(d). \tag{36}$$

Substituting for L(f) in (32), the time for any one evaluation of $\bar{A}^{(k)}$ or $\bar{A}^{(k)'}$ is dominated by

$$n^6 + n^4 L(d)^2. \tag{37}$$

During the isolation of the roots of $\bar{A}^{(k)}$, each complementary interval will be bisected at most $\le n^2 + nL(d)$ times. Since there are at most n complementary intervals and at most four evaluations are performed per bisection, the time for all evaluations necessary to isolate the roots of $\bar{A}^{(k)}$ is $\le n\{n^2+nL(d)\}\{n^6+n^4L(d)^2\} \sim n^9 + n^6 L(d)^3$. Since there are n polynomials $\bar{A}^{(k)}$, the total time is $\le n^{10} + n^7 L(d)^3$.

One must also consider the time required to refine the isolating intervals for the roots of $\bar{A}^{(k)}$ until the roots of $\bar{A}^{(k)'}$ are removed, using Algorithm R. By considering the, product $\bar{A}^{(k)}\bar{A}^{(k)'}$, and using (33) together with Mignotte's theorem, one finds that an interval containing a root of $\bar{A}^{(k)}$ and a different root of $\bar{A}^{(k)'}$ can not be shorter than $\frac{1}{2}(en^9 2^{8n} d^4)^n$ and it is then apparent that $n^{10} + n^7 L(d)^3$ also dominates the time for all Algorithm R applications.

Finally, we must consider the time required to compute $\bar{A}' = \mathrm{gsfd}(A')$ and $B = \gcd(A, \bar{A}')$, with A taking the values $\bar{A}^{(0)}, \bar{A}^{(1)}, \ldots, \bar{A}^{(n-2)}$. A modular polynomial greatest common divisor algorithm is used which, [2] has a maximum computing time dominated by $n^3 L(d)^2$ when applied to polynomials with degrees of n or less and norms of d or less. Since $L(|\bar{A}^{(k)}|_1) \le n + L(d)$, the time for all of these calculations is $\le n \cdot n^3\{n+L(d)\}^2 \sim n^6 + n^4 L(d)^2$. This completes the essential aspects of a proof of the following theorem.

Theorem 8. The computing time of the real root isolation algorithm described above when applied to the polynomial A with $\deg(A) = n$ and $|A|_1 = d$ is dominated by $n^{10} + n^7 L(d)^3$.

Heindel in [6] obtained a dominance bound of $n^{13} L(d)^3$ for his algorithm based on Sturm's theorem. However, he used a weaker theorem on the minimum root separation of a polynomial and a weaker theorem than Mignotte's to bound the norm of the greatest squarefree divisor of a polynomial. From [2] (pages 750-751), it is now easy to deduce that the time for Heindel's algorithm is also dominated by $n^{10} + n^7 L(d)^3$. Thus we are unable to prove that the maximum computing time of either algorithm is less than that of the other.

In the following section we will present numerous empirical comparisons of the two algorithms supporting the view that the derivative sequence algorithm is, in practice, generally significantly superior to the Sturm sequence algorithm. In practice, the roots of a polynomial are not nearly as close as the theoretical minimum. As a result, the required number of evaluations is correspondingly smaller and so are the numerators and denominators of the evaluation points. In fact, we will present data showing that, for several classes of polynomials, the observed number of evaluations is remarkably close to n^2 and quite independent of the norm d. Also, although we will not present the supporting data, it is surely no surprise that the observed lengths of the numerators and denominators depend very weakly on n and d. Probably the lengths grow slowly as a function of n, but so slowly that there is some justification for setting $L(e) \sim L(f) \sim 1$ in (29), yielding $n^2 + nL(d)$ as a hypothetical average computing time for one evaluation. This is the time for a polynomial of degree n and norm d, so to apply it to $\bar{A}^{(k)}$ we must substitute $L(2^n d) \sim n + L(d)$ for L(d); to within codominance the result remains $n^2 + nL(d)$. Now multiplying by n^2, the average number of evaluations, we obtain $n^4 + n^3 L(d)$ as a conjectured average time for all evaluations.

To this we must add the average time for computing the primitive derivative sequence. When we compute the content of $\bar{A}^{(k)'}$, the greatest common divisor of the first two coefficients will typically be n or less, thus single-precision. Thus we estimate time $\{n+L(d)\}^2 \sim n^2 + L(d)^2$ for this first greatest common divisor calculation and time $n + L(d)$ for each subsequent one. In this way we arrive at a conjectured average time of $n^3 + n^2 L(d)^2$ for computing the primitive derivative sequence.

We must further add an allowance for the time to compute $B_k = \mathrm{gsfd}(\bar{A}^{(k)'}) = \bar{A}^{(k)}/\gcd(\bar{A}^{(k)}, \bar{A}^{(k)'})$ and $\gcd(\bar{A}^{(k)}, B_k)$ for $0 \le k < n - 1$. We assume that for $k > 0$ these greatest common divisors will almost always be 1 and hence that the average time to compute each one will be dominated by $n\{n+L(d)\}$. But we do not assume that the input polynomial $A = \bar{A}^{(0)}$ is squarefree and therefore, in accordance with the discussion in [2], we allow $n^2 L(d) + nL(d)^2$ as the average time for $k = 0$. Thus the average computing time for all polynomial greatest common divisors is conjectured to be codominant with $n^2\{n+L(d)\} + \{n^2 L(d) + nL(d)^2\} \sim n^3 + nL(d)^2$.

We arrive therefore at a conjectured average computing time of $n^4 + n^3 + n^2L(d)^2$ $n^4 + n^2L(d)^2$ for the entire algorithm. Even this appears to be an over-estimate; the observed times presented in the next section indicate that when d is fixed the times are approximately proportional to n^3. The apparent discrepancy can perhaps be explained as follows. A large fraction of the total time is consumed by the evaluations, only a small fraction by the computation of the derivative sequence. The average time per evaluation is codominant with $n^2 + nL(d)$ and there are constants, c_1 and c_2, such that the average time per evaluation is reasonably well approximated by $c_1n^2 + c_2nL(d)$. But the constant c_1 is much smaller than c_2 so that, unless the ratio $n/L(d)$ is much larger than in the experiments which were performed, the term $c_2nL(d)$ predominates and the total time appears to be proportional to n^3 when d is fixed.

In contrast, the algorithm based on Sturm's theorem uses nearly all of its time on computing a Sturm sequence; very little time is spent evaluating the terms of the sequence. Since nearly all Sturm sequences are normal, the average time is dominated by $n^4L(d)^2$ and indeed the empirical evidence strongly supports the conjecture that the average time is codominant with $n^4L(d)^2$.

4. Empirical Comparisons

For convenience we will refer to the algorithm described in this paper as the DS algorithm (for derivative sequence) and to Heindel's algorithm based on Sturm's theorem as SS (for Sturm sequence). Both algorithms have been implemented within the SAC-1 computer algebra system and applied to a large number of polynomials of several kinds in order to obtain a representative comparison of performance. In the following we describe the polynomials and present the observed measurements. The computing times given are all measured in seconds. The computer used was the UNIVAC 1110 at the University of Wisconsin Madison Academic Computing Center. Actually, since this multiprogrammed computer has two central processors and two kinds of addressable memory with different speeds, the times presented are the fictitious times used for billing purposes. One "second" is really just $10^6/0.85$ memory references.

The first class of polynomials on which the algorithms were compared were random polynomials, that is, polynomials of specified degree n ~~with random integer coefficients of specified degree n~~ with random integer coefficients of specified bit length b, the coefficients being uniformly distributed in the open interval $(-2^b,+2^b)$. Each line of Table 1 gives the average outcome for three trials with the same specified degree and with b = 33. In this table and in subsequent tables, n denotes the degree, r the number of real roots, e the number of evaluations in the DS algorithm, t_1 the time for the DS algorithm, t_2 the time for the SS algorithm.

TABLE 1
Random Polynomials, b = 33

n	r	e	e/n^2	t_1	$10^3t_1/n^3$	t_2	$10^4t_2/n^4$
5	1	23	0.92	0.76	6.1	0.58	9.3
10	4/3	82	0.82	3.77	3.8	6.83	6.8
15	1	142	0.63	8.06	2.4	28.8	5.7
20	2	281	0.70	18.4	2.3	89.6	5.6
25	5/3	432	0.69	30.2	1.9	208.2	5.3

For random polynomials, Table 1 displays computing times approximately proportional to n^3 for the DS algorithm and n^4 for the SS algorithm, with DS about 7 times as fast when n = 25. The DS algorithm is especially advantageous when the coefficients are large. Table 2 shows, for n = 15, how slowly the computing time of DS increases as a function of the coefficient length b.

TABLE 2
Random Polynomials, n = 15

b	e	t_1
33	165	7.9
66	282	12.1
99	152	10.4
132	196	14.2

It is unlikely that random polynomials as defined above are typical of polynomials arising in real applications. For one thing, they have very few real roots, asymptotically an average of only $2\pi^{-1}\ln n$ (see [7], pages 5-12). The values of r above are in good agreement with this result. In the quantifier elimination algorithm [3] and in other applications, it is required to isolate the real roots of resultants. Therefore we generated resultants of random bivariate integral polynomials, chosen so that the resultants had degrees 8, 18 and 32, with coefficient lengths of approximately 10, 20 and 30 decimal digits. Surprisingly, the number of evaluations and computing time showed little dependency on the coefficient length, so the results were grouped by degree and averaged. Algorithm SS was not applied. Table 3 summarizes the observations.

TABLE 3
Random Resultants

n	r	e	e/n^2	t_1	$10^3t_1/n^3$
8	4	95	1.48	3.79	7.4
18	4	316	0.98	21.0	3.6
32	14/3	937	0.92	83.4	2.5

The random resultants have somewhat more real zeros than the random polynomials; the number of evaluations and the time are correspondingly slightly larger. Next we generated polynomials which were products of random linear polynomials. The coefficients of the linear polynomials were random 8-bit integers. After computing the product of n such linear polynomials, the bit length, say f, of the longest coeffient of the product was computed, and each coefficient a_i was then replaced by $[a_i/2^{f-33}]$. The resulting

integral polynomial of degree n, with longest coefficient 33 bits long, was then input to Algorithms DS and SS. The product, of course, has n real roots but the division of the coefficients perturbs the roots somewhat. After the perturbation, each polynomial continued to have n real roots except that in the case n = 20 two of the roots became complex. The results appear in Table 4.

TABLE 4
Random Products

n	e	e/n^2	t_1	$10^3 t_1/n^3$	t_2	$10^4 t_2/n^4$
5	55	2.20	1.42	11.4	0.54	8.7
10	205	2.05	6.83	6.8	6.47	6.5
15	483	2.15	17.9	5.3	29.1	5.8
20	836	2.09	38.0	4.8	89.7	5.6
25	1194	1.91	64.0	4.1	207.	5.4

The number of evaluations required for the random products was about three times the number required for the random polynomials, but the time required was only about twice as large. Perhaps this is because the points of evaluation were generally near roots and therefore the computed values were smaller. Notice that the times for the Sturm algorithm were nearly the same in each case.

Next the two algorithms were applied to the Chebyshev polynomials T_n of degree n, which may be defined by $T_0(x) = 1$, $T_1(x) = x$ and the recurrence relation $T_n(x) = 2xT_{n-1}(x) - T_{n-2}(x)$ for $n \geq 2$. As is well-known, T_n has n real roots. It is sufficient for present purposes to note that the bit length of the largest coefficient is between n and 4n/3. The results are shown in Table 5.

TABLE 5
Chebyshev Polynomials

n	e	e/n^2	t_1	$10^2 t_1/n^2$	t_2	$10^2 t_2/n^2$
5	43	1.72	0.78	3.1	0.37	1.5
10	157	1.57	3.40	3.4	1.84	1.8
15	345	1.53	8.98	4.0	4.90	2.2
20	583	1.46	18.6	4.6	10.8	2.7
25	910	1.46	35.2	5.6	19.9	3.2

Note that for the Chebyshev polynomials, the times for both algorithms are nearly proportional to n^2; $n^2 \log n$ would be a better approximation. Also, the Sturm sequence algorithm is nearly twice as fast. And this occurs in spite of the fact that the number of evaluations in the derivative sequence algorithm continues near n^2. The reason for this drastic change in behavior will be considered in a moment, but first we present, in Table 6, similar results for the Legendre polynomials P_n. The Legendre polynomials satisfy $P_0(x) = 1$, $P_1(x) = x$ and $(n+1)P_{n+1}(x) = (2n+1)xP_{n-1}(x) - nP_n(x)$ for $n \geq 1$. Thus defined, the coefficients are not integers, so we set $P_n^*(x) = 2^n P_n(x)$. We then have $P_0^*(x) = 1$, $P_1^*(x) = 2x$ and $(n+1)P_{n+1}^*(x) = (4n+2)xP_n^*(x) - 4nP_{n-1}^*(x)$ for $n \geq 1$. The coefficients of P_n^* are integers since $P_n^*(x) = (d/dx)^n(x^2-1)^n/n!$ For our purposes it suffices to note that the largest coefficient of P_n^* is approximately 2n bits long. Although the Legendre polynomial coefficients are appreciably longer, both algorithms perform almost the same as for the Chebyshev polynomials.

TABLE 6
Legendre Polynomials

n	e	e/n^2	t_1	$10^2 t_1/n^2$	t_2	$10^2 t_2/n^2$
5	39	1.56	0.74	3.0	0.42	1.7
10	149	1.49	3.31	3.3	1.70	1.7
15	320	1.42	8.40	3.7	4.72	2.1
20	555	1.39	17.8	4.4	10.7	2.7
25	877	1.40	34.8	5.6	22.4	3.6

The Sturm sequence algorithm is much faster for Chebyshev and Legendre polynomial than for random polynomials or random products--about 10 times as fast for degree 25--whereas the derivative sequence algorithm performs at pretty much the same speed on all four classes (although it is somewhat slower on random products). The explanation is derived from the observation that the Chebyshev and Legendre polynomials have Sturm sequences with exceptionally small coefficients. Table 7 shows, for degree 25, the bit length of the largest coefficient of each term in the primitive Sturm sequence for each of the four types of polynomials: R (random polynomial), P (random product), C (Chebyshev), and L (Legendre). At the bottom of the table the times are given for computing each Sturm sequence.

The primitive derivative sequences of the Chebyshev polynomials also have unusually small coefficients. But since primitive derivative sequences have relatively small coefficients in general, the effect is much less pronounced. This is shown in Table 8, which has the same structure as Table 7.

We do not know how to explain the unusual behavior of the Sturm sequences of Chebyshev and Legendre polynomials. The only other polynomials known by us to have Sturm sequences with unexpectedly small coefficients are the Stirling polynomials $S_n(x) = x(x-1)\ldots(x-n+1)$. This was noted by George W. Reitwiesner in 1972 in a letter to Collins.

5. Conclusions

We have described and analyzed in considerable depth a new algorithm for isolating the real roots of a polynomial. This algorithm uses the sequence of derivatives of the given polynomial, relies on Rolle's theorem, and uses a tangent construction as necessary to decide whether a given interval contains two roots or none. We have compared the new algorithm, both analytically and empirically with a Sturm sequence algorithm and found the new algorithm quite generally and appreciably faster. The primary cause of the improved performance is the fact that derivative sequences generally have much smaller

TABLE 7
Primitive Sturm Sequences

Term	R	P	C	L
1	33	33	29	49
2	37	36	29	45
3	73	52	25	43
4	131	86	26	42
5	201	127	24	37
6	272	176	24	36
7	340	228	21	34
8	407	281	21	33
9	475	336	19	30
10	547	394	19	29
11	619	448	14	27
12	689	498	17	26
13	758	548	14	21
14	828	604	14	19
15	899	656	11	17
16	974	707	12	16
17	1045	754	10	13
18	1118	795	9	12
19	1188	837	5	10
20	1254	875	7	9
21	1326	911	5	6
22	1397	952	5	5
23	1466	986	2	3
24	1531	1012	3	3
25	1595	1037	1	1
26	1	1	1	1
Time	200.4	108.1	2.33	2.99

TABLE 8
Primitive Derivative Sequences

Term	R	P	C	L
1	33	33	29	49
2	38	36	29	45
3	41	39	29	41
4	44	41	32	38
5	46	43	33	33
6	49	44	34	35
7	50	45	31	30
8	52	46	35	30
9	53	46	31	31
10	54	46	30	31
11	54	46	24	28
12	55	46	26	26
13	55	45	22	23
14	55	44	22	27
15	55	43	18	23
16	54	42	19	22
17	54	40	16	17
18	51	39	18	16
19	52	36	15	10
20	50	34	15	13
21	49	29	12	12
22	46	25	10	12
23	44	24	5	6
24	41	21	6	6
25	37	16	1	1
26	1	1	1	1
Time	2.18	1.90	1.09	1.13

coefficients than Sturm sequences, and this also results in a significant reduction in the amount of memory required. A further reduction in required memory is achieved by recapturing derivatives through integration. The new algorithm has an even greater advantage in those applications where it is also required to obtain isolating intervals for the real roots of any derivative of the given polynomial, because they are in any case computed.

References

[1] Collins, G.E., The SAC-1 Integer Arithmetic System--Version III, Technical Report WIS-CS-156-73, Comp. Sci. Dept., Univ. of Wisconsin, March 1973.

[2] Collins, G.E., Computer Algebra of Polynomials and Rational Functions, Am. Math. Monthly, Vol. 80, No. 7 (Aug.-Sept. 1973), pp. 725-755.

[3] Collins, G.E., Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition, Lecture Notes in Computer Science, Vol. 33, pp. 134-183, Springer Verlag, Berlin, 1975.

[4] Collins, G.E., High-precision Calculation of Real Algebraic Numbers, in preparation.

[5] Collins, G.E., and Horowitz, E., The Minimum Root Separation of a Polynomial, Mathematics of Computation, Vol. 28, No. 126 (April, 1974), pp. 589-597.

[6] Heindel, L.E., Integer Arithmetic Algorithms for Polynomial Real Zero Determination, Journal ACM, Vol. 18, No. 4 (Oct., 1971),

[7] Kac, M., Probability and Related Topics in Physical Sciences, Interscience Publishers, New York, 1959.

[8] Knuth, D.E., The Art of Computer Programming, Vol. 2 (Seminumerical Algorithms), Addison-Wesley, Reading, Mass., 1968.

[9] Mignotte, M., An Inequality About Factors of Polynomials, Mathematics of Computation, Vol. 28, No. 128 (October, 1974), pp. 1153-1157.

[10] van der Waerden, B. L., Modern Algebra, Ungar Publishing Co., New York, 1948.

On Square-Free Decomposition Algorithms

by

David Y. Y. Yun
Mathematical Sciences Department
IBM Thomas J. Watson Research Center
Yorktown Heights, N. Y.

1. Introduction

Given a polynomial P in R[x] where R is a unique factorization domain, P is said to be square-free if P has no divisor (or factor) of multiplicity ≥ 2. The problem of finding the square-free (abbreviated as SQFR) decomposition of P amounts to finding polynomials $P_1, P_2, \ldots, P_k$ such that $P = P_1 P_2^2 \ldots P_k^k$, $P_k \neq 1$, each P_i is square-free, and gcd $(P_i, P_j)=1$, $i \neq j \leq k$.

Square-free decompositions of polynomials have many uses. Among them, the most important ones are in polynomial factorization and partial fraction decomposition of rational functions. These are, in turn, indispensable tools of rational function integrations [MOS67], [MOS71], [HOR71], [RIS69]. In addition, the SQFR decomposition provides a useful and inexpensive alternative toward the computation of multiplicative basis in the radical simplification problem [FAT72], [C&F76]. Horowitz [HOR71] made a quite complete survey and analysis on computing square-free decompositions, partial fraction decompositions and rational function integrations for the univariate case. Many existing symbolic manipulation systems such as MACSYMA [MAC73], SAC-1 [HOR69], and SCRATCHPAD [GJY75] also include built-in functions for computing square-free decompositions of multivariate polynomials. Musser [MUS71] presented an improved version of the SQFR decomposition algorithm by Horowitz. In this paper, we will give three different (including revamped versions of Horowitz's and Musser's) algorithms for computing the SQFR decomposition of polynomials in R[x]. Some algorithm analysis will be carried out to show the (asymptotic) superiority of the algorithm we propose (last of the three). The basic idea for this algorithm is, however, surprisingly simple and can be easily realized to be computationally more efficient. Finally, two versions of multivariate SQFR decomposition algorithms using the principles of modular arithmetic and the Hensel Construction will be presented. One of them stands out as it takes the full advantage of all the techniques. Both versions do establish that the cost of computing the SQFR decomposition is not much more than that of computing the greatest common divisor (GCD) of the polynomial and its derivative.

2. Three Direct Algorithms

First we state and prove a fundamental fact:

Theorem 1: If $P = P_1 P_2^2 \ldots P_k^k$ is primitive in x then $\gcd(P, dP/dx) = P_2 P_3^2 \ldots P_k^{k-1}$.

Proof: Let $D = \gcd(P, dP/dx)$. For an arbitrary P_i write $P = P_i^i Q$, then $dP/dx = i\, P_i^{i-1}(dP_i/dx)\, Q + P_i^i (dQ/dx)$, i.e., P_i^{i-1} divides dP/dx. Thus $\prod_{i=2}^{k} P_i^{i-1}$

divides D since the P_i's are pairwise relatively prime. On the other hand, $dP/dx = \left(\prod_{i=2}^{k} P_i^{i-1}\right)\left(\sum_{i=1}^{k} i(dP_i/dx) \prod_{j\neq i} P_j\right)$ Let S be a divisor of some P_j, then S does not divide any other P_i for $i\neq j$ since $\gcd(P_i,P_j)= 1$ for all $i\neq j$. Because P_j is square-free and primitive w.r.t. x, $\gcd(P_j,dP_j/dx) = 1$, so that S does not divide dP_j/dx. That means S divides every term in the summation except one-- $j(dP_j/dx)$, hence S does not divide the sum.

We now present three square-free decomposition algorithms where given a primitive polynomial P(x) in R[x], the algorithms will compute square-free and pairwise relatively prime polynomials $P_i(x)$, i = 1,...,k, (some of them may equal 1) such that $P = P_1P_2^2 \ldots P_k^k$, for some $k \leq 1$ and $P_k\neq 1$.

(a) (Tobey and Horowitz):

$C_1 \leftarrow \gcd(P, dP/dx);$

$D_1 \leftarrow P/C_1;$

For i = 1, step 1, until $C_i = 1$, do:

$C_{i+1} \leftarrow \gcd(C_i, dC_i/dx);$

$D_{i+1} \leftarrow C_i/C_{i+1};$

$P_i \leftarrow D_i/D_{i+1}.$

$C_1 = P_2P_3^2 \ldots P_k^{k-1}$

$D_1 = P_1P_2 \ldots P_k$

$C_i = P_{i+1} P_{i+2}^2 \ldots P_k^{k-1}$

$D_i = P_iP_{i+1} \ldots P_k$

$C_{i+1} = P_{i+2} P_{i+3}^2 \ldots P_k^{k-i-1}$

$D_{i+1} = P_{i+1} \ldots P_k$

(b) (Musser):

$C_1 \leftarrow \gcd(P, dP/dx);$

$D_1 \leftarrow P/C_1;$

For i = 1, step 1, until $D_i = 1$, do:

$D_{i+1} \leftarrow \gcd(C_i,D_i);$

$C_{i+1} \leftarrow C_i/D_{i+1};$

$P_i \leftarrow D_i/D_{i+1};$

$C_1 = P_2P_3^2 \ldots P_k^{k-1}$

$D_1 = P_1 \ldots P_k$

$C_i = P_{i+1} P_{i+2}^2 \ldots P_k^{k-1};$

$D_i = P_i \ldots P_k$

$D_{i+1} = P_{i+1} \ldots P_k$

$C_{i+1} = P_{i+2} P_{i+3}^2 \ldots P_k^{k-i-1}$

(c) $G \leftarrow \gcd(P, dP/dx)$

$C_1 \leftarrow P/G$

$D_1 \leftarrow (dP/dx)/G - DC_1/dx$

For i = 1, step 1, until $C_i = 1$, do:

$P_i = \gcd(C_i,D_i);$

$C_{i+1} = C_i/P_i;$

$D_{i+1} = D_i/P_i - dC_{i+1}/dx.$

$G = P_2P_3^2 \ldots P_k^{k-1}$

$C_1 = P_1 \ldots P_k$

$D_1 = P_1 \frac{dP_2}{dx} \ldots P_k + 2P_1P_2 \frac{dP_3}{dx} \ldots P_k$

$+ \ldots +(k-1)\, P_1P_2 \ldots \frac{dP_k}{dx}$

$C_i = P_iP_{i+1} \ldots P_k$

$D_i = P_i(\frac{dP_{i+1}}{dx} \ldots P_k + 2P_{i+1} \frac{dP_{i+2}}{dx} \ldots P_k$

$+ \ldots +(k-i)\, P_{i+1} \ldots \frac{dp_k}{dx})$

To show the validity of Algorithm (c), essentially by induction, we need to establish two facts, both are corollaries of Theorem 1.

Corollary 1.1: Let $P'=dP/dx$, $G=\gcd(P,P')$, then

$$P'/G-(P/G)' = P_1(P_2'P_3 \ldots P_k + 2P_2P_3' \ldots P_k + \ldots + (k-1)\, P_2P_3 \ldots P_{k-1}P_k').$$

Proof. By the chain rule:

$$P' = P_1'P_2^2 \ldots P_k^k + P_1 2P_2'P_2\, P_3^3 \ldots P_k^k + \ldots + P_1P_2^2 \ldots P_{k-1}^{k-1}\, k\, P_k'\, P_k^{k-1}.$$

$$P'/G = P_1'P_2 \ldots P_k + 2P_1P_2'P_3 \ldots P_k + \ldots + k\, P_1P_2 \ldots P_{k-1}\, P_k'.$$

$P/G = P_1P_2 \ldots P_k$, so

$$(P/G)' = P_1'P_2 \ldots P_k + P_1\, P_2'\, P_3 \ldots P_k + \ldots + P_1 \ldots P_{k-1}\, P_k'.$$

Therefore $(P'/G) - (P/G)'$ is as claimed. //

Corollary 1.2: $\gcd(P/G,\ (P'/G) - (P/G)') = P_1$

Proof. $P/G = P_1 \ldots P_k$ and $(P'/G)-(P/G)'=P_1Q$ where

$$Q = P_2'P_3 \ldots P_k + 2P_2\, P_3'\, P_4 \ldots P_k + \ldots + (k-1)\, P_2 \ldots P_{k-1}\, P_k'.$$

Since P_1 divides both, P_1 divides, or is a divisor of, the GCD. To see that P_1 is the GCD, it suffices to show that $\gcd(P_2 \ldots P_k, Q) = 1$. Let $P^* = P_2P_3^2 \ldots P_k^{k-1}$, then Theorem 1 provides that $G^* = \gcd(P^*,dP^*/dx) = P_3P_4^2 \ldots P_k^{k-2}$ where the corresponding cofactors are relatively prime, i.e., $\gcd(P^*/G^*,\ (dP^*/dx)/G^*) = 1$. But $P^*/G^* = P_2 \ldots P_k$ and $(dP^*/dx)/G^* = Q$. //

3. Comparison of Algorithms

For Algorithms (a) and (b), $D_i=P_iP_{i+1} \ldots P_k$ and $C_i = P_{i+1}P_{i+2}^2 \ldots P_k^{k-1}$ are computed in different ways. Algorithm (b) appears to be more efficient for the following reasons: (1) the GCD computations involve smaller inputs for Algorithm (b), since D_i is obviously smaller than dC_i/dx; (2) Each P_i is computed in Algorithms (a) via one more division than by Algorithm (b).

Computation of GCD's often has the ultimate goal of cancelling the GCD and obtain the simpler cofactors of the corresponding input polynomials (as in the case of simplifying a rational number or rational function. Thus, it is often convenient as well as efficient to compute all three useful outputs through the call of one function. In the case of the Modular GCD Algorithm as stated and analyzed by Brown [BRO71], the cofactors are essentially by-products of the GCD computation, hence the cost of getting all three outputs is the same as that of getting the GCD alone. With such an algorithm in mind, we readily see that Algorithm (b) wastes no operations at all as opposed to Algorithm (a). Similarly, Algorithm (c) also utilizes all three outputs of the GCD computations.

Let the operation $(G,A^*,B^*) \leftarrow \gcd(A,B)$ be consisting of a call to a GCD computing function which also produces the corresponding cofactors and a parallel assignment of G to the GCD, A* to A/G, and B* to B/G. Then the above three algorithms can be restated in more concise forms as follows:

(a) $(C_1,D_1,W) \leftarrow \gcd(P,dP/dx)$;
For i=1, step 1, until $C_i = 1$, do:
$(C_{i+1},D_{i+1},W) \leftarrow \gcd(C_i,dC_i/dx)$;
$P_i \leftarrow D_i/D_{i+1}$.

(b) $(C_1,D_1,W) \leftarrow \gcd(P,dP/dx)$;
For i=1, step 1, until $D_i=1$, do:
$(D_{i+1},C_{i+1},P_i) \leftarrow \gcd(C_i,D_i)$.

(c) $(W,C_1,D_1) \leftarrow \gcd(P,dP/dx)$;
For i=1, step 1, until $C_i=1$, do:
$(P_i,C_{i+1},D_{i+1}) \leftarrow \gcd(C_i,D_i-dC_i/dx)$.

With these reformulated algorithms, it is easier to see the extraneous efforts involved in Algorithm (a). "W" denotes unused output of GCD computations and Algorithm (a) has one such waste

at each iteration. In addition, there is also one extra division for each iteration of Algorithm (a).

Both Algorithms (b) and (c) have no wasted outputs from GCD computations within the "do-Loops". As a rough comparison between the two algorithms, we again compare the sizes (e.g., degrees) of the inputs to the GCD calculations. For Algorithm (b), the inputs are $C_i = P_{i+1}P_{i+2}^2 \cdots P_k^{k-1}$ and $D_i = P_i \cdots P_k$ while for Algorithm (c), they are $C_i = P_iP_{i+1} \cdots P_k$ and $D_i = P_i\left(\frac{dP_{i+1}}{dx} \cdots P_k + 2P_{i+1}\frac{dP_{i+2}}{dx} \cdots P_k + \cdots + (k-i)P_{i+1}\cdots\frac{dP_k}{dx}\right)$ where the degree of this D_i is one less than the degree of C_i. To give a more detailed comparison of these two algorithms, we will consider that the Modular GCD Algorithm [BRO71] is used which has a computing cost of

$$\ell^2(d+1)^v + \ell(d+1)^{v+1}$$

where ℓ is the maximum number of digits in the coefficient and d is the maximum degree in all v variables of the two input polynomials.

It is also necessary to make some assumptions on the sizes of the given polynomial P and the desired answers. Let $P = P_1P_2^2 \cdots P_k^k$ where $\deg(P_i) = d$, for $i = 1,2,\ldots,k$. Then $\deg(P) = dk(k+1)/2$. Assume also that N is the coefficient bound for, or the largest coefficient in P_i for all i. Then the length of N or the number of bits in N is $n = \log N$. For any j^{th} power of P_i, its size bounds in terms of coefficient length and degree are jn and jd, respectively.

For both Algorithms (b) and (c), C_i is the larger of the two inputs to the GCD computation at each iteration, so the total cost for the algorithms is

$$\text{Cost} = \sum_{i=1}^{k} [\text{length}(C_i)^2 (\deg(C_i)+1)^v + \text{length}(C_i)(\deg(C_i)+1)^{v+1}].$$

As we shall see, the univariate case will already show a difference in the computing costs of Algorithms (b) and (c). So let v = 1,

$$\text{Cost} \geq \sum_{i=1}^{k} \text{length}(C_i)^2\deg(C_i) + \sum_{i=1}^{k} \text{length}(C_i)\deg(C_i)^2.$$

The input polynomials to the GCD computations for the loop in Algorithm (b) are C_i and D_i. It is easy to see that C_i is usually the larger polynomial in terms of both degree, $\sum_{j=i+1}^{k} jd = d(k-i)(k+i+1)/2$, and coefficient length $\sum_{j=i+1}^{k} jn = n(k-i)(k+i+1)/2$. Using these bounds on sizes, we get

$$\text{Cost(b)} = \sum_{i=1}^{k} \frac{(n(k-i)(k+i+1))^2}{2} \cdot \frac{d(k-i)(k+i+1)}{2} + \sum_{i=1}^{k} \frac{nd^2(k-i)^3(k+i+1)^3}{8} = (n^2d + nd^2) \cdot \frac{k(k-1)(2k-1)(12k^4 - 24k^3 + 21k^2 - 9k+4)}{420}.$$

Correspondingly, the inputs to the GCD computations in Algorithm (c) are C_i and D_i where $\deg(D_i) = \deg(C_i) - 1$. Since $C_i = P_iP_{i+1} \cdots P_k$, $\deg(C_i) = (k-i+1)d$ and $\text{length}(C_i) = (k-i+1)n$. We then get

$$\text{Cost(c)} = \sum_{i=1}^{k} (n(k-i+1))^2 (d(k-i+1)) = \sum_{i=1}^{k} (n(k-i+1))(d(k-i+1))^2 = (n^2d+nd^2)\frac{k^2(k+1)^2}{4}.$$

Roughly speaking, cost (b) is approximately $O((2/35)k^7(n^2d + nd^2))$ as compared to cost(c) which is $O((1/4)k^4(n^2d + nd^2))$.

The computing cost for the Modular GCD Algorithm is slightly different in case the two input polynomials are relatively prime, i.e., GCD = 1. A formula of $\ell^2(d+1)^v+(d+1)^{v+1}$ is given by Brown [BRO71]. In view of that, we point out that while no GCD in the loop of Algorithm (b) is 1 until the last, as many GCD's as there are P_i's that are 1 can be equal to 1 in Algorithm (c). Thus we briefly show the difference in the computing costs of

these two algorithms for a special case where P_i's are equal 1 and have sizes (n,d) alternatingly. Let $P = P_1 P_i^2 \ldots P_k^k$ where deg $(P_i) = d$, for all $i = 2j$, $j = 1,2,\ldots,\ell$, $k = 2\ell$, and $P_i = 1$, for all $i = 2j-1$, $j = 1,2,\ldots,\ell$. Thus $\deg(P) = \sum_{j=1}^{\ell} 2jd = d\ell(\ell+1)$.

For Algorithm (b), again C_i is larger than D_i where

$$\deg(C_i) = \begin{cases} \sum_{j=\frac{i+1}{2}}^{\ell} (2j-1)d = (\ell^2 - (\frac{i-1}{2})^2)d & \text{for i odd} \\ \sum_{j=i/2}^{\ell} 2jd = (\ell+i/2)(\ell+i/2+1) & \text{for i even} \end{cases}$$

So

$$\text{Cost (b)} = (n^2d + nd^2)\left(\sum_{j=1}^{\ell} (\ell^2-(j-1)^2)^3 + \sum_{j=1}^{\ell} (\ell+j)^3 (\ell+j+1)^3 \right)$$

$$= (n^2d + nd^2) \frac{\ell(d+1)(186\ell^5 + 449\ell^4 + 357\ell^3 + 93\ell^2 + \ell - 1)}{10}$$

$$= O((93/640)\, k^7 (n^2d + nd^2)).$$

For Algorithm (c), $\deg(C_i) = \deg(D_i) + 1$ or

$$\deg(C_i) = \begin{cases} (\ell - \frac{i+1}{2} + 1)\, d & \text{for i odd} \\ (\ell - \frac{i}{2} + 1)\, d & \text{for i even.} \end{cases}$$

So

$$\text{Cost(c)} = \sum_{j=1}^{\ell} n^2 d(\ell-j+1)^3 + \sum_{j=1}^{\ell} d^2(\ell-j+1)^2 \quad (\text{GCD} = 1)$$

$$+ (n^2d+nd^2) \sum_{j=1}^{\ell} (\ell-j+1)^3 \quad (\text{GCD} \neq 1)$$

$$= (2n^2d+nd^2) \frac{\ell^2(\ell+1)^2}{4} + \frac{d^2\ell(\ell+1)(2\ell+1)}{6}$$

$$= O((1/32)k^4n^2d + (1/64)k^4nd^2 + (1/24)k^3d^2).$$

Note that due to all the rounding-off these comparisons of computing cost formulas do not show realistic differences until the parameter k becomes large. Some empirical testing bears this fact out such that the actual computing times for Algorithm (c), although always stays below that of Algorithm (b), do not show drastic differences for small values of k(up to k = 6).

4. Multivariate SQFR Algorithms

Yun proposed and implemented a square-free decomposition algorithm for multivariate polynomials [YUN74] [MAC73] which is based on a generalization of the Hensel's lifting technique called the Multivariate EZ Algorithm [YUN74], [M&Y73], [W&R75]. This SQFR algorithm, called EZSQFR Algorithm, parallels very much the outline of the EZGCD Algorithm of Moses and Yun [M&Y73]. It uses the technique of evaluation homomorphisms and the Hensel-type or p-adic constructions [YUN74] [YUN75] in much the same way as Brown's Modular GCD Algorithm uses homomorphisms and the Chinese Remainder Algorithm. We will not discuss these techniques in detail for the lack of space, but only refer interested readers to the above references.

Before presenting the multivariate EZSQFR Algorithm we will first discuss a useful, time-saving device for the computation of square-free decompositions, due to a suggestion by P.Wang. A given multivariate polynomial could already be square-free with respect to a particular variable. If this fact cam be detected, at a relatively small cost, before the entire machinery of the square-free algorithm begins to work, there can be big savings in computing time. There is such a time saving test which we shall call fail-safe square-free test (f.s.s.f.). This test consists of (a) evaluating the polynomial at random by chosen "valid" ([YUN74] or see † below) points. (b)

computing the GCD of the resulting univariate polynomial and its derivative w.r.t. the main variable. (c) checking to see if the GCD is an integer or not, if so, then the original polynomial is square-free. The validity of this test is shown by the following lemma:

Lemma 2: Let $b = (b_1,\ldots,b_v)$ be an arbitrary valid evaluation for $P(x,y_1,\ldots,y_v)$ in $Z[x,y_1,\ldots,y_v]$. If $P_b(x) = P(x,b_1,\ldots,b_v)$ is square-free in $Z(x)$ then P is itself square-free.

Proof: If P is not square-free, $P = QR^2$, then via the evaluation homomorphism $P_b = Q_b R_b^2$ which cannot be square-free. //

The usefulness of this test comes from the next lemma, which is also stated and proved by Wang and Rothschild [W&R75]:

Lemma 3: If P is a square-free multivariate polynomial in $Z[x,y_1,\ldots,y_v]$, then a set of integers $b = (b_1,\ldots,b_v)$ can be chosen so that be is a valid evaluation for P(i.e., † deg(P)=deg($P_b(x)$)) and $P_b(x)$ is also square-free.

Proof: Let $P = P_1P_2 \ldots P_k$ be the factorization of P into irreducible factors where $\gcd(P_i,P_j) = 1$ for all i and j such that $i \neq j$. $P_b(x)$ is square-free if and only if $P_{i_b}(x)$ is square-free for all i and $\gcd(P_{i_b}(x),P_{i_b}(x))$ = constant, for all $i \neq j$, which is equivalent to $reslt(P_{i_b}(x),P_{j_b}(x)) \neq 0$, for all i,j, $i \neq j$, where "reslt" denotes the resultant w.r.t. x[VDW49]. Let $R(x,y_1,\ldots,y_v) = \prod_i reslt(P_i,dP_i/dx) \prod_{i<j} reslt(P_i,P_j)$. Then $R \neq 0$ since P is square-free and $P_b(x)$ is square-free if and only if $R_b(x) \neq 0$. Now it suffices to note that there are only finitely many integral values for $b_1,\ldots,b_v$ such that $R_b(x) = 0$ or $(lc(P))_b = 0$. //

The finiteness of integral roots for $R(x,y_1,\ldots,y_v)$ has the further implication that out of the infinite possibilities of integral values for each b, the probability for choosing a valid but unlucky set b, such that $P_b(x)$ becomes not square-free when P is, cannot be too large. We will apply this test at the beginning of the EZSQFR Algorithm with at most two valid evaluations. If the test fails, we simply assume no information was gained and continue with the rest of the EZSQFR Algorithm. However, we point out that not all the work done for the test was wasted in this case, since the computations done for the f.s.s.f. test is useful for the EZGCD Algorithm when it is used to compute the square-free part of the given polynomial.

Next, we will prove a lemma which will eliminate the need for using the special case method of the EZGCD Algorithm [M&Y73] when it is invoked to compute N = gcd(P, dP/dx) at the second step of the EZSQFR Algorithm.

Lemma 4: Let P be primitive in $(Z[y_1,\ldots,y_v])[x]$ and R = gcd(P, dP/dx), then gcd(R, (dP/dx)/R) = 1.

Proof: Let $P = P_1P_2^2 \ldots P_k^k$ and Q = dP/dx. By Theorem 1, $R = P_2^1 P_3^2 \ldots P_k^{k-1}$. Assume D = gcd(R, Q/R) ≠ 1, then Q = R D Q" for some Q". Since D divides R, there must be an irreducible factor C ≠ 1 of some P_i, $i \geq 2$. But this C clearly also divides P/R so that P = R C P" for some P". Thus, R C divides both P and Q, contradicting to R being the GCD. Therefore, gcd(R, Q/R) must be 1. //

This lemma and the finiteness of the number of unlucky evaluations imply that the polynomial dP/dx can always be used for the application of the p-adic construction. Also, the special case

method of the EZGCD Algorithm can be completely avoided for this square-free decomposition algorithm. We are ready now to describe the EZSQFR Algorithm in detail. However, we will not be so careful as to indicate, for example, how the computations for f.s.s.f test is used in the EZGCD Algorithm. If the given polynomial is not primitive, then clearly we can work on its content and primitive part separately. Because of the fact that square-free decompositions are main variable dependent and that for some uses it is not necessary to square-free decompose the content, we will assume primitive inputs to the EZSQFR Algorithm.

Algorithm (d): (EZSQFR)

Input: A primitive multivariate polynomial $P(x,y_1,\ldots,y_v)$ in $Z[x,y_1,\ldots,y_v]$ and the main variables, x.

Output: A list of polynomials in $Z[x,y_1,\ldots,y_v]$, $P_1,P_2,\ldots,P_k$, such that $P = P_1P_2^2 \ldots P_k^k$ for some $k \geq 1$ where $\gcd(P_i,dP_i/dx) = 1$, $\gcd(P_i,P_j) = 1$ for $i \neq j$, and $P_k \neq 1$.

(1) Invoke the f.s.s.f. test twice with random valid evaluations. If the test is successful, then return P, otherwise continue.

(2) Apply the EZGCD Algorithm on P and dP/dx to obtain $N = \gcd(P,dP/dx)$ and $L = P/N$. Let b be the valid lucky evaluation for P and p be the lucky prime for $P_b(x)$ used in the EZGCD Algorithm above.

(3) Set $L_b \leftarrow L(x,b_1,\ldots,b_v)$, $N_b \leftarrow N(x,b_1,\ldots,b_v)$, and $i \leftarrow 1$.

(4) Apply UNIGCO Algorithm (or some other univariate GCD Algorithm which also computes cofactors) on L_b and N_b to obtain $H_o \leftarrow \gcd(L_b, N_b)$, $G_o \leftarrow L_b/H_o$, and $N_b \leftarrow N_b/H_o$. Set $L_b \leftarrow H_o$.

(5) If $G_o = 1$, then set $P_i \leftarrow 1$. Otherwise, apply the Multivariate EZ Algorithm on L, G_o, and H_o to get multivariate polynomials G and H such that $L = G\,H$ over Z. Set $P_i \leftarrow G$, $L \leftarrow H$, and $i \leftarrow i + 1$.

(6) If $N_b = 1$, then go to (4). Otherwise, set $k \leftarrow 1$, $P_k \leftarrow L$, and return $P_1, P_2,\ldots, P_k$.

Remark: Note that even the evaluation of multivariate polynomial L and N in Step (3) need only be done once. The later values for L_b and N_b are simply updated with known quantities. Step (4) involves only one univariate GCD computation. For the application of the Multivariate EZ Algorithm in Step (5), the required lucky evaluation b and prime p is again provided by the EZGCD Algorithm in Step (2). Thus the results of that computation are certain to be correct over Z.

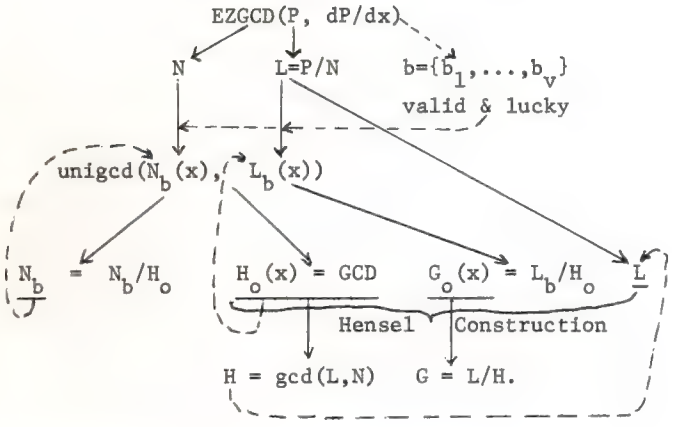
The EZSQFR Algorithm takes great advantages of computations already performed to achieve added efficiencies. By performing Hensel constructions instead of full-fledged multivariate GCD computations, the gains in efficiency are clear. Even the preparations for these applications of the Multivariate EZ Algorithm are very simple. They involve essentially only one univariate GCD computation as preparation for each Hensel construction, hence for each P_i. That is much less costly than what EZGCD Algorithm goes through for preparing Hensel construction. All the choosing of lucky evaluations and primes, testing for various conditions etc., are eliminated because the evaluation values and the prime are known to be lucky after the first call to EZGCD.

Whenever $P_i = 1$ for some i (for example, $P = U\,V^3$, then $P_1 = U, P_2 = 1, P_3 = V$), the computational process is even simpler - only one uni-

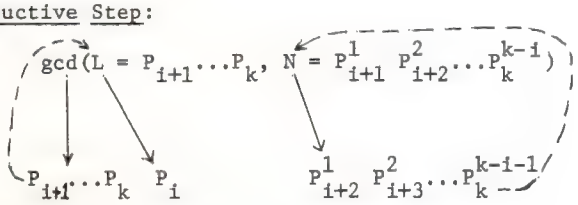
variate GCD computation is necessary and no multivariate operation is required at all. Compared to Algorithm (b), we would find in this case, $D_{i+1} = \gcd(C_i, D_i) = D_i$, so that $P_i = D_i/D_{i+1} = 1$, i.e. D_i divides C_i. Even so, there is still at least one multivariate division involved, which is more time consuming than one univariate GCD computation in most cases.

Algorithm (d), EZSQFR, as presented in detail above clearly is closely related to Algorithm (b). For the GCD computations in the loop, both algorithms use as inputs the remaining square-free part and the non-square-free part, i.e., Algorithm (b) uses $D_i = P_i \cdots P_k$ and $C_i = P_{i+1} P_{i+2}^2 \cdots P_k^{k-i}$ whereas Algorithm (d) uses the univariate images, $L_b(x)$ and $N_b(x)$, of these. To make the flow of ideas clearer, we present a schematic diagram of the EZSQFR Algorithm (d), together with a short concise version of the Algorithm.

Schematic Diagram of EZSQFR



Inductive Step:



Algorithm (d): EZSQFR (applying the Hensel Construction)

(i) $(N, L, W) \leftarrow$ EZGCD $(P, dP/dx)$.

(ii) Use the valid and lucky evaluation $b = \{b_1, \ldots, b_v\}$ for P from above EZGCD computation to compute $L_b(x) = L(x, b_1, \ldots, b_v)$ and $N_b(x) = N(x, b_1, \ldots, b_v)$. Set $i \leftarrow 1$.

(iii) Loop: $(L_b, G_0, N_b) \leftarrow$ unigcd $(L_b(x), N_b(x))$ (univariate GCD computation).
If $G_0 = 1$, then set $P_i \leftarrow 1$, otherwise apply the Hensel Construction on L, G_0, $H_0 \leftarrow L_b$ to get G and H such that $L = GH$. Set $P_i \leftarrow G$, $L \leftarrow H$, and $i \leftarrow i+1$.
If $N_b(x) \neq 1$, go Loop, else set $P_i \leftarrow L$ and return $P_1, P_2, \ldots, P_k$ where $k = i$.

Corresponding to Algorithm (c), it is obviously also possible to extend and generalize that to the multivariate case and apply the Hensel or Multivariate EZ Algorithm. We present such a version only in short form as Algorithm (e) and give a schematic diagram as follows

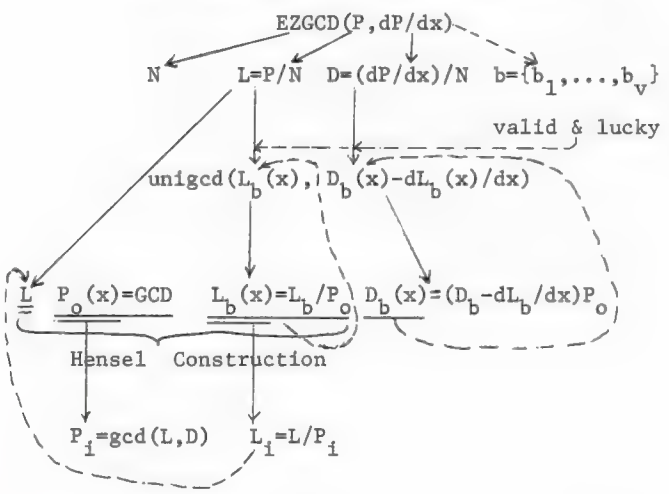
Algorithm (e):

(i) $(N, L, D) \leftarrow$ EZGCD $(P, dP/dx)$

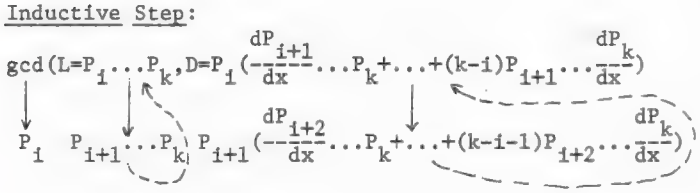
(ii) Use the valid and lucky evaluation points $b = \{b_1, \ldots, b_v\}$ for P from the above EZGCD computation to get $L_b(x) = L(x, b_1, \ldots, b_v)$ and $D_b(x) = D(x, b_1, \ldots, b_v)$. Set $i \leftarrow 1$.

(iii) Loop:
$(P_0, L_b, D_b) \leftarrow$ unigcd $(L_b(x), D_b(x) - dL_b(x)/dx)$.
If $P_0 = 1$, then set $P_i \leftarrow 1$, otherwise apply the Hensel Construction on L, P_0, L_b to get P_i and $L_i = P_i L_i$. Set $L \leftarrow L_i$ and $i \leftarrow i+1$.
If $L \neq 1$, go Loop, else set $P_i \leftarrow L$ and return $P_1, P_2, \ldots, P_k$ where $k \leftarrow i$.

Schematic Diagram of Algorithm(e)



Inductive Step:



In summary, both Algorithm (d) and (e), while being generalizations of Algorithms (b) and (c), respectively, show that the cost for computing square-free decompositions of multivariate polynomials is essentially the same as the cost of computing GCD's. To see that, we need only realize that all the Hensel Construction applications in the loops of both algorithms amount to the work of lifting the relatively prime factors of $L = P_1 P_2 \ldots P_k$. However, that effort is dominated by the cost of lifting L and D = (dP/dx)/L from dP/dx in the Hensel Construction stage of the EZGCD computation. All the other work in both algorithms is comparatively negligible. Therefore, the cost of EZGCD(P, dP/dx) is the dominant cost for Algorithms (d) and (e).

5. Conclusions

In this paper, we presented three algorithms for directly computing the square-free decomposition of polynomials by GCD calculations. Two of them, Algorithm (a) and (b), have been known. They are included here for completeness and comparison. Algorithm (c) involves surprisingly simple ideas, yet it quite obviously constitutes an improvement over the other two. Further improvements are realized as we consider the advantages of Modular GCD Algorithms. Several comments by Brown [BRO71] on the efficiency of the modular approach to GCD computations are actually put in practice here: (1) Modular GCD Algorithm computes the GCD and the corresponding cofactors essentially at no extra cost; (2) In terms of dense polynomial arithmetic, the cost of Modular GCD Algorithm is comparable to that of one division of the same polynomial inputs; (3) Modular approach to GCD computation achieves added efficiency when the input polynomials are relatively prime. The advantages resulting from all these points on algorithm efficiency are taken by Algorithm (c), as demonstrated by our analysis and comparison of these three algorithms.

Without losing the advantages of the modular approach, the Hensel or p-adic construction is brought into the computation of SQFR decompositions, especially in the multivariate case. Algorithms (b) and (c) are extended respectively to Algorithms (d) (i.e. EZSQFR) and (e). While preserving their original efficiencies, they showed the new feature that the cost of computing the SQFR decomposition of a polynomial is comparable (related by a constant factor) to that of computing the GCD of the same polynomial and its derivative.

Based on that conclusion, we feel the SQFR decomposition can be effectively used as an improvement to the factored representation of rational functions as proposed by Brown [BRO74] and Hall [HAL74]. Since their method obtains factors by

GCD computations while avoiding the expensive process of complete factorization, the SQFR factored representation will provide more factors also at the expense of GCD computations. Furthermore the SQFR representation of polynomials is canonical which is not the case for the partially factored numerator and denominator of rational functions in their scheme. Therefore, the possibility of using the square-free canonical representation for polynomials in some future algebraic systems (at a cost not much more than a GCD computation) is enhanced and suggested.

REFERENCES

[BRO71] Brown, W.S., "On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors", JACM, Vol. 18, No. 4, October 1971, pp. 478-504.

[BRO74] Brown, W.S., "On Computing with Factored Rational Expressions", Proceedings of EUROSAM '74, (ACM SIGSAM Bulletin No. 31), pp. 27-34.

[C&F76] Caviness, B.F., and Fateman, R.J., "Simplication of Radical Expressions", these proceedings.

[FAT72] Fateman, R.J., "Essays in Algebraic Simplification", Ph.D Thesis, Harvard, also MAC TR-95, MIT, April 1972.

[GJY75] Griesmer, J.H., Jenks, R.D., and Yun, D.Y.Y., "SCRATCHPAD User's Manual", IBM T.J. Watson Research Center Report - RA70, June 1975.

[HAL74] Hall, A.D., "Factored Rational Expressions in ALTRAN", Proceeding of EUROSAM '74, pp. 35-45.

[HEN13] Hensel, K., Zahlentheorie, Goschen, Berlin and Leipzig, 1913.

[HOR69] Horowitz, E., "Algorithms for Symbolic Integration of Rational Functions", Ph.D. Thesis, Computer Science Dept., University of Wisconsin, 1969.

[HOR71] Horowitz, E., "Algorithms for Partial Fraction Decomposition and Rational Function Integration", [PET71], pp. 441-457.

[KNU69] Knuth, D.E., The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, Addison Wesley, Reading, Mass. 1969.

[MAC73] MACSYMA User's Manual, Bogen, R., et al., Project MAC, M.I.T., Cambridge, Mass.,1973.

[MOS67] Moses, J., "Symbolic Integration", MAC-TR-47, Project MAC, M.I.T., Dec., 1967.

[MOS71] Moses, J., "Symbolic Integration: The Stormy Decade", [PET71], March 1971, pp. 427-440.

[MUS71] Musser, D.R., "Algorithms for Polynomial Factorization", Ph.D. Thesis, C.S. Dept., Univ. of Wisconsin, August 1971.

[M&Y73] Moses, J. and Yun, D.Y.Y., "The EZGCD Algorithm", Proc. of ACM Annual Conference, Aug. 1973, Atlanta, pp. 159-166.

[M&Y74] Miola, A. and Yun, D.Y.Y., "The Computational Aspects of Hensel-type Univariate Polynomial Greatest Common Divisor Algorithms", Proc. EUROSAM '74, Aug. 1974, pp. 46-54.

[PET71] Proc. of the Second Symposium on Symbolic and Algebraic Manipulation, Petrick, S.R., ed., ACM, March 1971.

[RIS69] Risch, R., "The Problem of Integration in Finite Terms", Trans. AMS, Vol. 139, May 1969, pp. 167-189.

[TOB67] Tobey, R.G., "Algorithms for Antidifferentiation of Rational Functions", Ph.D. Thesis, Harvard, 1967.

[W&R75] Wang, P.S. and Rothschild, L.P., "Factoring Multivariate Polynomials over the Integers", Math. of Comp. Vol. 29, No. 131, July 1975, pp. 935-950.

[YUN74] Yun, D.Y.Y., "The Hensel Lemma in Algebraic Manipulation", Ph.D. Thesis, Dept. of Math. MAC-TR-138, M.I.T., Nov. 1974.

[YUN75] Yun, D.Y.Y., "Hensel Meets Newton -- Algebraic Construction in an Analytic Setting", Analytic Computational Complexity, Proc. of CMU Symposium, Traub, J., ed., Academic Press, April 1975.

[YUN76] Yun, D.Y.Y., "P-adic Constructions and its Applications in Algebraic Manipulation", these proceedings.

Algorithms for Gaussian Integer Arithmetic[1]

by

B. F. Caviness,[2] Computer Science Department, Illinois Institute of Technology, Chicago, Illinois 60616

and

G. E. Collins, Computer Sciences Department, University of Wisconsin, 1210 W. Dayton Street, Madison, Wisconsin 53706

Abstract

In this paper new algorithms are given for Gaussian integer division and the calculation of the greatest common divisor of two Gaussian integers. Empirical tests show that the new gcd algorithm is up to 5.39 times as fast as a Euclidean algorithm using the new division algorithm.

1. Introduction

A Gaussian integer is a number of the form a_1+a_2i where a_1 and a_2 are (rational) integers and $i = \sqrt{-1}$. The Gaussian integers constitute a ring, denoted by G, in which addition and multiplication are defined by

$$(a_1+a_2i)+(b_1+b_2i) = (a_1+b_1)+(a_2+b_2)i, \qquad (1)$$

$$(a_1+a_2i)(b_1+b_2i) = (a_1b_1-a_2b_2)+(a_1b_2+a_2b_1)i. \qquad (2)$$

Algorithms for addition and multiplication in G are therefore trivial when based on the use of formulas (1) and (2) and the corresponding algorithms for arithmetic in Z, the ring of rational integers. However, we will see that for multiplication of large Gaussian integers the use of (2) is not quite optimal.

It is easy to verify that G is an integral domain so that division is defined, and some interesting algorithmic questions arise in connection with division in spite of the well-known classical formula

$$(a_1+a_2i)/(b_1+b_2i) = (a_1b_1+a_2b_2)/(b_1^2+b_2^2) + (a_2b_1-a_1b_2)i/(b_1^2+b_2^2) \qquad (3)$$

resulting from multiplying both (a_1+a_2i) and (b_1+b_2i) by $b_1 - b_2i$, the conjugate of $b_1 + b_2i$. A new division algorithm is discussed in section 2.

G is also a Euclidean domain and hence a g.c.d. domain. In section 3 a new algorithm for greatest common divisor calculations in G is given. Empirical tests show that the new algorithm is approximately five times as fast as the Euclidean algorithm implemented using the division algorithm of section 2.

We define the length of the Gaussian integer $a = a_1 + a_2i$ by the formula

$$L(a_1+a_2i) = L(a_1) + L(a_2) \qquad (4)$$

where the length of an integer a_1, is given by $L(a_1) = \lfloor \log_\beta |a_1| \rfloor + 1$, for $a_1 \neq 0$. $L(a_1)$ is the number of β-radix digits in the integer a_1. If $a_1 = 0$, $L(a_1) = 1$. If the base β is important, we write $L_\beta(a_1)$ instead of $L(a_1)$.

a_1 and a_2 are respectively called the real part, denoted rp(a), and the imaginary part, denoted ip(a), of a.

Let t(a+b) and t(a-b) respectively denote the time to compute the sum and difference of the Gaussian integers a and b. Equation (1) gives algorithms with $t(a+b) \leq L(a) + L(b)$ and $t(a-b) \leq L(a) + L(b)$. (See [Col 71] for definitions of the dominance relations $\leq$ and $\sim$.)

Equation (2) provides an algorithm for Gaussian integer multiplication involving four multiplications and two additions or subtractions of integers. The number of multiplications can be reduced to three by noticing that

$$a_1b_2 + a_2b_1 = (a_1+a_2)(b_1+b_2) - a_1b_1 - a_2b_2 \qquad (5)$$

or that

$$a_1b_1 - a_2b_2 = a_1(b_1+b_2) - b_2(a_1+a_2), \qquad (6)$$

$$a_1b_2 + a_2b_1 = a_2(b_1-b_2) + b_2(a_1+a_2). \qquad (7)$$

The number of additions and subtractions is increased to five by these two methods.

The use of either (5), or (6) and (7) with the classical algorithm for multiplication of integers, reduces the computing time for Gaussian integer multiplication in SAC-1 by nearly 25% whenever

[1] This research was supported in part by National Science Foundation Grants GJ-30125X and DCR74-13278.

[2] Present address: Mathematical Sciences Department, Rensselaer Polytechnic Institute, Troy, N. Y. 12181

$L(a_1) = L(a_2) = L(b_1) = L(b_2) = n$ and n is sufficiently large. When n is small the computing time will be increased somewhat by the extra additions and subtractions, which will take longer than the saved multiplication. Comparisons have been made for SAC-1 algorithms based on (2) and (6)-(7) for $L(a_1) = L(a_2) = m$ and $L(b_1) = L(b_2) = n$ for small values of m and n. The experimental results showed that the latter algorithm never requires more than 10% more time than the former. When $m = n$ the latter algorithm is faster when $n \geq 4$, 11% faster when $n = 8$, and 22% faster for $n = 16$.

The computing times for the algorithms analyzed herein depend, directly or indirectly, on the computing time of integer multiplication. We will express these times in terms of an unspecified function M(n) where the time required to multiply two integers, both of length n or less, is dominated by M(n). If one is interested in theoretically fast algorithms M(n) can be considered to be $nL(n)L(L(n))$, the computing time for the Schönhage-Strassen algorithm [ScS 71]. On the other hand if one is interested in the computing time of a practical algorithm for relatively small integers, it may be best to use the classical multiplication algorithm for which $M(n)$ is n^2.

Given two integers a_1, a_2 of lengths m and n respectively it is possible to design a multiplication algorithm whose computing time is dominated by the function

$$M'(m,n) = \begin{cases} mn^{-1}M(n) & \text{if } m \geq n \\ nm^{-1}M(m) & \text{if } n > m. \end{cases}$$

The basic idea behind such an algorithm is the breaking-up of a_1 (assume without loss of generality that $m \geq n$) into pieces each of length n and then multiplying each piece by a_2. Henceforth we shall assume that we have such an algorithm at our disposal.

The time to multiply two Gaussian integers $a = a_1 + a_2 i$ and $b = b_1 + b_2 i$ is dominated by $M'(L(a),L(b))$ since each multiplication required in either of the three algorithms discussed above is dominated by $M'(L(a),L(b))$.

In order to prove the desired computing time theorems we make the following reasonable assumptions about the function M(n). For real numbers x and y we assume that

$$\left.\begin{array}{l} 0 < M(x) < M(y) \\ 0 < \dfrac{M(x)}{x} < \dfrac{M(y)}{y} \end{array}\right\} \quad \text{for } 0 < x < y. \qquad (8)$$

Furthermore we assume $x \leq M(x) \leq x^2$ and that $M(cx)/cx \leq M(x)/x$ for all real numbers $c > 0$.

From (8) we have that for $x > 0$

$$M'(x,y) < M'(x,z) \text{ for } 0 < y < z. \qquad (9)$$

The conjugate of $a = a_1 + a_2 i$ is $a_1 - a_2 i$, denoted by conj(a) or $\bar{a}$. The product of a and its conjugate is the non-negative integer $a_1^2 + a_2^2$, called the norm of a and is denoted by norm(a) or N(a). The times to compute conj(a) and N(a) are dominated by m = L(a) and M(m) respectively.

In order to analyze the computing time of the algorithms we need the following:

Lemma 1. Let a and b be Gaussian integers. Then $L(a) \sim L(N(a))$ and if a and b are non-zero, $L(a \cdot b) \sim L(a) + L(b)$.

Proof. The proof of the first part is easy. To prove the second part use the fact that $N(a \cdot b) = N(a) \cdot N(b)$ and then apply the first part. ∎

The units of G are $1 = i^0$, i, $-1 = i^2$ and $-i = i^3$. If $a = a_1 + a_2 i$ is any non-zero complex number, a_1 and a_2 real, we will say that a is in the first quadrant in case $a_1 > 0$ and $a_2 \geq 0$, in the second quadrant if $a_1 \leq 0$ and $a_2 > 0$, in the third if $a_1 < 0$ and $a_2 \leq 0$, in the fourth if $a_1 \geq 0$ and $a_2 < 0$. Clearly any non-zero complex number is in exactly one quadrant and if a is in the j-th quadrant, $j > 1$, then $i^{5-j}a$ is in the first quadrant. Hence every non-zero Gaussian integer a has a unique first-quadrant associate, denoted by fqa(a). 0 is the only associate of 0 and we set fqa(0) = 0.

The first-quadrant associate of a Gaussian integer a can be computed in time dominated by L(a).

2. Division

Now let us consider division of Gaussian integers. According to (3), the quotient of two Gaussian integers is not in general a Gaussian integer but is instead a Gaussian rational, that is, a number of the form $r_1 + r_2 i$ where r_1 and r_2 are ordinary rational numbers. Accordingly, given a and b, $b \neq 0$, we are to compute a quotient q and remainder r, both Gaussian integers, such that $a = bq + r$. Here we must of course add some additional conditions so that q and r will be uniquely determined. Since the most important application of division with remainder is the Euclidean algorithm, we certainly want to require that $|r| < |b|$ or equivalently, $N(r) = |r|^2 < |b|^2 = N(b)$, but this still does not uniquely determine q and r.

If $a_1 + a_2 i$ is any complex number, a_1 and a_2 real, then we define $\{a_1 + a_2 i\}$, the nearest Gaussian integer to $a_1 + a_2 i$, by

$$\{a_1 + a_2 i\} = \{a_1\} + \{a_2\} i.$$

where $\{a\} = \lceil a - 1/2 \rceil$ when a is real. We then have

$$N(a - \{a\}) \leq 1/2$$

for any complex number a.

If a and b are Gaussian integers and $q = \{\frac{a}{b}\}$ then q is the nearest quotient of a and b and $r = a - bq$ is the least remainder of a and b, denoted rem(a,b). The latter terminology is justified by observing that if $\bar{q}$ is the Gaussian rational a/b and $a = bq' + r'$ then $\bar{q} - q' = r'/b$ and $\bar{q} - q = r/b$ so $|r| \leq |r'|$. Notice also that $|r'| < |b|$ if and only if $|\bar{q} - q'| < 1$.

We easily obtain an algorithm for the nearest quotient and least remainder of Gaussian integers using formula (3) and the corresponding algorithm for ordinary integers.

GINQR1(A,B;Q,R)

[Gaussian integer nearest quotient and remainder, first algorithm. A and B are Gaussian integers, $B \neq 0$. $Q = \{A/B\}$ and $R = A - BQ$.]
(1) [A = 0?] If A = 0 then (Q←0; R←0; return).
(2) [Compute $A\cdot\bar{B}$ and N(B).] $\bar{B} \leftarrow$ conj(B); $C \leftarrow A\cdot\bar{B}$; $D \leftarrow$ norm(B).
(3) [Compute nearest quotient and remainder for real and imaginary parts.] $C_1 \leftarrow$ rp(C); $C_2 \leftarrow$ ip(C); $Q_1 \leftarrow \{C_1/D\}$; $R_1 \leftarrow C_1 - DQ_1$; $Q_2 \leftarrow \{C_2/D\}$; $R_2 \leftarrow C_2 - DQ_2$; $Q \leftarrow Q_1 + Q_2 i$; $R \leftarrow R_1 + R_2 i$. ■

Theorem 2. The time to compute Q and R is $\leq M(L(B)) + M'(L(B),L(Q))$.

Proof. Let $k = L(A)$, $m = L(B)$, $n = L(Q)$. The time to execute step (2) $\leq M(m) + M'(k,m) \leq M(m) + M'(m,n)$. To see that $M(m) + M'(k,m) \leq M(m) + M'(m,n)$ consider first the case $m \leq n$. Then $M'(m,n) = mn^{-1}M(m)$. Now $M'(k,m)$ is either $km^{-1}M(m)$ or $mk^{-1}M(k)$. But since $k \sim m + n$, $km^{-1}M(m) \sim (m+n)m^{-1}M(m) \sim nm^{-1}M(m) = M'(m,n)$. On the other hand $M'(k,m) = mk^{-1}M(k)$ only if $k < m$ which in combination with $m \leq n$ and (8) implies that $M'(k,m) = mk^{-1}M(k) \leq nm^{-1}M(m) = M'(m,n)$ and the desired result follows.

Otherwise assume $m > n$ which implies that $k \sim m + n \sim m$ and that $\max(k,m) \sim m$, $\min(k,m) \sim m$ and $M'(k,m) \sim mm^{-1}M(m) = M(m)$ and thus $M(m) + M'(k,m) \leq M(m) \leq M(m) + M'(m,n)$.

Using a division algorithm based on Newton's algorithm (see [AHU 74] Section 8.2 for the basic ideas) Q_1, R_1, and Q_2, R_2 can be computed in time dominated by $M'(L(Q_1), L(D))$ and $M'(L(Q_2), L(D))$ respectively. But $M'(L(Q_1), L(D)) + M'(L(Q_2), L(D)) \leq M'(L(Q), L(B))$ from which the theorem follows. ■

The algorithm GINQR1 is not entirely satisfactory because of the term $M(L(B))$ in Theorem 2. When $L(B) \leq L(Q)$, $M(L(B)) \leq M'(L(B),L(Q))$ and the $M(L(B))$ term is unimportant. But when the quotient Q is small relative to the divisor B, the term $M(L(B))$, representing the time to compute the norm of B, is dominant. It would be interesting to know whether there is an algorithm to compute the nearest quotient and least remainder of two Gaussian integers in time dominated by $M'(L(B), L(Q))$, so that the time for division of Gaussian integers would be analogous to that for rational integers. Such an algorithm has not been discovered and it seems doubtful that one exists.

Here we present an algorithm that avoids computing norm (B) in a large percentage of cases where norm(B) > norm(Q). The main idea is to compute an approximation to the Gaussian rational $Q = A/B$ using approximations A* and B* to A and B having only a few more bits than $Q = \{A/B\}$. This is possible because if $Q = Q_1 + Q_2 i$ we can estimate in advance the maximum of $L_2(Q_1)$ and $L_2(Q_2)$, the number of bits in Q_1 and Q_2 respectively.

First we present the mathematical justification for the algorithm and then the algorithm which is called GINQR. For any rational integer a, define

$$L_2^*(a) = \begin{cases} L_2(a), & \text{if } a \neq 0 \\ 0, & \text{if } a = 0 \end{cases} \tag{10}$$

(Recall that $L(0) = L_2(0) = 1$.) Clearly the time to compute $L_2^*(a)$ is $\sim L_2(a) \sim L(a)$. If now $a = a_1 + a_2 i$ is any Gaussian integer, define

$$L_2^*(a_1+a_2 i) = \max\,(L_2^*(a_1), L_2^*(a_2)). \tag{11}$$

The time to compute $L_2^*(a)$ for a in G is also dominated by L(a).

Let $m_i = L_2^*(a_i)$. If $a_i \neq 0$ then $2^{m_i-1} \leq a_i < 2^{m_i}$ and hence $2^{2m_i-2} \leq a_i^2 < 2^{2m_i}$. So if $a \neq 0$ then

$$2^{2m-2} \leq N(a) < 2^{2m+1}, \tag{12}$$

where $m = L_2^*(a)$. Hence also,

$$2^{m-1} \leq |a| < 2^{m+1/2}. \tag{13}$$

Now let b be another non-zero Gaussian integer with $n = L_2^*(b)$. Then by (13),

$$2^{m-n-3/2} < |a/b| < 2^{m-n+3/2}. \tag{14}$$

If $m \geq n$ then $|a/b - \{a/b\}| \leq 2^{-1/2}$ so $|\{a/b\}| < 2^{m-n+3/2} + 2^{-1/2} \leq 2^{m-n}(2^{3/2}+2^{-1/2}) < 2^{m-n+2}$. If $m - n \leq -1$ then $|a/b| < 2^{1/2} < 3/2$ so $|\{a/b\}| \leq 1$. If $m - n \leq -3$ then $|a/b| < 2^{-3/2} < 1/2$ so $\{a/b\} = 0$. Hence

$$|\{a/b\}| \leq 2^{m-n+2} \tag{15}$$

for all m and n. In particular,

$$\{a/b\} = 0 \tag{16}$$

for $m - n \leq -3$. Similarly, if $m - n \geq 3$, then $|\{a/b\}| > 2^{m-n-3/2} - 2^{-1/2} \geq 2^{m-n-3}(2^{3/2}-2^{-1/2}) > 2^{m-n-2}$. If $m - n = 2$ then $|a/b| > 2^{1/2}$ so

$|\{a/b\}| \geq 1$. If $m - n = 1$ then $|a/b| > 2^{-1/2}$; hence if $a/b = \bar{q} = \bar{q}_1 + \bar{q}_2 i$ then $\bar{q}_1^2 + \bar{q}_2^2 > 1/2$, $\max(\bar{q}_1^2, \bar{q}_2^2) > 1/4$, $\max(|\bar{q}_1|, |\bar{q}_2|) > 1/2$ and $\{a/b\} \neq 0$. So

$$|\{a/b\}| \geq 2^{m-n-2} \tag{17}$$

for $m - n \geq 1$.

Let $\ell = m - n$. By (16), $q = \{a/b\} = 0$ when $\ell \leq -3$. By (15) and (17), we have otherwise $\ell - 2 \leq L_2^*(q) \leq \ell + 3$. Let k be a fixed small positive integer the precise value of which will be chosen later and let $h = \ell + k$. If $h \geq n$, i.e., if the quotient is large relative to the divisor, we will apply GINQR1 to a and b to compute q, but if $h < n$ we will apply GINQR1 to a* and b* where a* and b* are respectively 2h-bit and h-bit approximations to a and b. More precisely, for any complex number $a = a_1 + a_2 i$, define

$$[a_1 + a_2 i] = [a_1] + [a_2]\, i, \tag{18}$$

called the integral part of a. Since $L_2(a) = m$ and $L_2(b) = n$, we set

$$a^* = [a/2^{m-2h}], \tag{19}$$

$$b^* = [b/2^{n-h}]. \tag{20}$$

We then have $L_2(a^*) = 2h$ and $L_2(b^*) = h$.

$a^* = [a/2^m]$, $0 \leq m < L_2(a)$, can be computed in time $\leq L(a)$.

By (19) and (20), if GINQR1 is applied to a* and b*, a quotient q* will be produced with $q^* \simeq (a/b)2^{n-m+h} = \bar{q}2^k$. If k is sufficiently large then $q^*/2^k$ will be a very good approximation to $\bar{q}$ and, unless $\bar{q}_1 - 1/2$ or $\bar{q}_2 - 1/2$ is very close to an integer, we will have $q = \{q^*/2^k\}$. Let us now examine the accuracy of this approximation.

For any complex numbers a_1, a_2, ε_1 and ε_2 with $a_2 \neq 0$ and $\varepsilon_2 \neq -1$, we have

$$\left|\frac{a_1(1+\varepsilon_1)}{a_2(1+\varepsilon_2)} - \frac{a_1}{a_2}\right| = \left|\frac{a_1}{a_2}\right| \cdot \left|\frac{\varepsilon_1 - \varepsilon_2}{1+\varepsilon_2}\right| \leq \left|\frac{a_1}{a_2}\right| \cdot \frac{|\varepsilon_1| + |\varepsilon_2|}{1-|\varepsilon_2|}. \tag{21}$$

In (21) let us set $a_1 = a/2^{m-2h}$, $a_1(1+\varepsilon_1) = a^*$, $a_2 = b/2^{n-h}$, $a_2(1+\varepsilon_2) = b^*$. Then by (19), $|a_1\varepsilon_1| < 1$ so by (13), $|\varepsilon_1| < 1/|a_1| = 2^{m-2h}/|a| \leq 2^{1-2h}$. Similarly by (20) and (13), $|\varepsilon_2| < 2^{1-h}$. $q = 0$ when $\ell = m - n \leq -3$, so we may assume that $\ell \geq -2$. We will assume now also that $k \geq 6$. Then $h = \ell + k \geq 4$ so $|\varepsilon_2| < 1/8$ and $(1-|\varepsilon_2|)^{-1} < 8/7$. Also,

$$|\varepsilon_1| + |\varepsilon_2| < 2^{1-2h} + 2^{1-h} = 2^{1-h}(1+2^{-h}) \leq \frac{17}{16} \cdot 2^{1-h}$$

so $(|\varepsilon_1|+|\varepsilon_2|)(1-|\varepsilon_2|)^{-1} < \frac{8}{7} \cdot \frac{17}{16} \cdot 2^{1-h} = \frac{17}{7} \cdot 2^{-h}$. Thus by (21),

$$\left|\frac{a^*}{b^*} - \frac{a}{b} \cdot 2^k\right| < \frac{17}{7} \cdot 2^{-h}. \tag{22}$$

Since $q^* = \{a^*/b^*\}$, $|q^* - a^*/b^*| \leq 1/\sqrt{2}$. Hence by (22), since $h \geq 4$, $|q^* - a2^k/b| < 1/\sqrt{2} + 17/112 < 1$. Dividing by 2^k,

$$|q^*2^{-k} - \bar{q}| < 2^{-k}. \tag{23}$$

Thus, unless $|q_1 - \{\bar{q}_1\}| \geq 1/2 - 2^{-k}$ or $|\bar{q}_2 - \{\bar{q}_2\}| \geq 1/2 - 2^{-k}$, we will have $q = \{\bar{q}\} = \{q^*/2^k\}$. Figure 1 shows a unit square in the complex plane with Gaussian integers as vertices, and $q = \{q^*/2^k\}$ except when $\bar{q}$ is in the shaded regions. Thus $q \neq \{q^*/2^k\}$ with probability less than 2^{-k+2}.

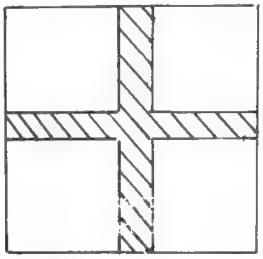


Figure 1.
The probability that $q = \{q^*/2^k\}$.

Let $c_i = \{q_i^*/2^k\}$ and $d_i = q_i^* - c_i 2^k$, so that $|d_i| \leq 2^{k-1}$. By (23), $c_i = \{\bar{q}_i\} = q_i$ unless $|d_i| = 2^{k-1}$. So if $|d_i| \neq 2^{k-1}$ for $i = 1,2$ then q is determined and the remainder r can be computed as $a - bq$. Otherwise, the process just described can be repeated with 2k in place of k. in general, k will be doubled as often as necessary until $h = \ell + k \geq n$, and then GINQR1 will be applied to a and b. We will use 10 as the initial value of k, but any initial value not less than 6 could be used.

In the algorithm description below capital letters denote variables that can take on multiple-precision values and small letters denote single-precision variables.

GINQR(A,B;Q,R)

[Gaussian integer quotient and remainder. A and B are Gaussian integers, $B \neq 0$. $Q = \{A/B\}$ and $R = A - BQ$.]

(1) [A = 0?] if A = 0 then (Q←R←0; return).
(2) [Compute lengths.] $m \leftarrow L_2^*(A)$; $n \leftarrow L_2^*(B)$; $\ell \leftarrow m - n$.
(3) [$\ell \leq -3$?] if $\ell \leq -3$ then (Q←0; R←A; return).
(4) [Initialize k.] $k \leftarrow 10$.

(5) [$h \geq n$?] $h \leftarrow \ell + k$; if $h \geq n$ then (GINQR1 (A,B;Q,R); return).

(6) [Compute Q*.] $A^* \leftarrow [A/2^{m-2h}]$; $B^* \leftarrow [B/2^{n-h}]$; GINQR1(A*,B*; Q*,R*).

(7) [Compute Q and R.] $Q_1^* \leftarrow rp(Q^*)$; $Q_2^* \leftarrow ip(Q^*)$; $Q_1 \leftarrow \{Q_1^*/2^k\}$; $R_1 \leftarrow Q_1^* - 2^k Q_1$; if $L_2^*(R_1) = k$ then goto (8); $Q_2 \leftarrow \{Q_2^*/2^k\}$; $R_2 \leftarrow Q_2^* - 2^k Q_2$; if $L_2^*(R_2) = k$ then goto (8); $Q \leftarrow Q_1 + Q_2 i$; $R \leftarrow A - BQ$; return.

(8) [Double k.] $k \leftarrow 2k$; goto (5). ■

Theorem 3. The maximum computing time for GINQR applied to inputs A and $B \leq L(m_1)M(m_1)$ and the average computing time $\leq M'(m_1,m_2)$ where $Q = \{A/B\}$, $m_1 = \max(L(B),L(Q))$ and $m_2 = \min(L(B),L(Q))$.

Proof. Let $m = L_2^*(A)$, $n = L_2^*(B)$, $\ell = m - n$. For any k the time to execute steps (5) - (8), excluding the last execution of step (5) and the last execution of step (7), is $\leq m_1 + M(\ell+k)$. The successive values of k are $k_0 = 10$, $2k_0$, $4k_0,\ldots,2^j k_0$, where $\ell + 2^j k_0 < n$. Hence, steps (5) - (8) are executed at most $L_2(n) \leq L(m)$ times and the time for each execution, except perhaps the last, is $\leq m_1 + M(n) \leq M(m_1)$. But the time for the execution of steps (1) to (4) and the last execution of step (5) or (7) is $\leq M(m_1)$ so the maximum computing time is $\leq L(m_1)M(m_1)$.

By our previous remark, steps (5) - (8) are executed j+1 times with probability $\leq 2^{-jk_0+2}$. Hence the average execution time of steps (5) - (8) excluding the final application of GINQR1 or the time to compute the product B·Q in step (7), is dominated by

$$M(\ell+k_0) + 2^{-k_0} M(\ell+2k_0) + 2^{-2k_0} M(\ell+4k_0) + \ldots$$

$$\leq M(\ell+k_0) + 2^{2-k_0} M(\ell+k_0) + 2^{4-2k_0} M(\ell+k_0) + \ldots$$

$$\leq M(\ell+k_0) \sum_{i=0}^{\infty} (2^{2-k_0})^i = M(\ell+k_0)(1-2^{2-k_0})^{-1}$$

$$< M(\ell+k_0).$$

If $\ell + k_0 \geq n$ then GINQR1 is applied to A and B immediately in step (5), so assume $\ell + k_0 < n$. Then $\ell < n$ so $\ell + k_0 \sim L(Q) \sim \min(L(B),L(Q)) = m_2$ by (15) and (17). Hence the average execution time is $\leq M(\ell+k_0) + M'(m_1,m_2) \sim M(m_2) + M'(m_1,m_2) \sim M'(m_1,m_2)$. ■

3. Greatest Common Divisor Calculation.

The problem of computing greatest common divisors of Gaussian integers has many similarities with the problem for ordinary integers, but there are also several difficulties which must be surmounted. Perhaps these difficulties can be summarized by observing that the various analyses and theorems relating to greatest common divisor calculation for integers rely heavily on the order relation in Z, the ring of integers, whereas G is not an ordered domain.

Let a and b be arbitrary non-zero Gaussian integers. Let $a_1, a_2, \ldots, a_{r+1}$ by any sequence of Gaussian integers such that $a_1 = a$, $a_2 = b$, $|a_1| \geq |a_2| > |a_3| > \ldots > |a_r| > |a_{r+1}| = 0$, $r \geq 2$, such that for $1 \leq i \leq r-1$ there is a Gaussian integer q_i such that

$$a_i = q_i a_{i+1} + a_{i+2}. \tag{24}$$

$a_1,a_2,\ldots,a_{r+1}$ will then be called a remainder sequence (of a and b), and $q_1,\ldots,q_{r-1}$ is its associated quotient sequence. Such a remainder sequence always exists because we can set

$$q_i = \{a_i/a_{i+1}\} \tag{25}$$

and then determine a_{i+2} by (24). Dividing (24) by a_{i+1} we obtain $a_{i+2}/a_{i+1} = a_i/a_{i+1} - q_i = a_i/a_{i+1} - \{a_i/a_{i+1}\}$, so $|a_{i+2}/a_{i+1}| \leq |1/2+(1/2)i| = 1/\sqrt{2}$ and

$$|a_{i+2}| \leq (1/\sqrt{2})|a_{i+1}|. \tag{26}$$

The unique remainder sequence so determined will be called the least-remainder sequence of a and b. It is easy to see that if $a_1,a_2,\ldots,a_{r+1}$ is any remainder sequence of a and b then a_r is a greatest common divisor of a and b. That greatest common divisor of a and b in the first quadrant will be called the greatest common divisor of a and b and denoted by gcd(a,b). If $a \neq 0$ then gcd(a,0) = gcd(0,a) is the first quadrant associate of a, and gcd(0,0) = 0. Associated with any remainder sequence $a_1,a_2,\ldots,a_{r+1}$ are unique first and second cosequences defined by

$$u_1 = 1,\ u_2 = 0,\ u_i = q_i u_{i+1} + u_{i+2}, \tag{27}$$

$$v_1 = 0,\ v_2 = 1,\ v_i = q_i v_{i+1} + v_{i+2}. \tag{28}$$

Using (27), (28) and induction we have that

$$u_i a + v_i b = a_i \tag{29}$$

for all i.

In this section (29) is used along with a fast method for computing $q_1,q_2,\ldots,q_i,q_{i+1}$ and u_i,u_{i+1},v_i,v_{i+1} to obtain a_i and a_{i+1} and hence obtain a fast gcd algorithm for Gaussian integers.

The well-known Lehmer algorithm [Leh 38] for the calculation of greatest common divisors of integers can be generalized in a more or less straightforward way to Gaussian integers through the use of exact complex-interval arithmetic. In fact such an algorithm has been programmed in SAC-1 and was discussed by Caviness [Cav 73]. Empirical

tests with the SAC-1 implementation of the generalized Lehmer algorithm showed that it is in fact up to 4.7 times as fast as the Euclidean algorithm using (3) for the division of Gaussian integers. A straightforward algorithm was used for division of complex intervals in the implementation of the Lehmer algorithm and this algorithm gave, in general, intervals that were much larger than was desirable and hence slowed down the Lehmer algorithm. In an effort to improve this aspect of the algorithm the currently presented algorithm was discovered. This algorithm is about twice as fast as the Lehmer algorithm, does not use complex-interval arithmetic and hence is simpler, but it does use the idea of the original Lehmer algorithm of approximating the sequence of quotients by using the high-order, single-precision parts of the inputs.

The algorithm uses a single-precision Euclidean algorithm when the remainders are small enough. We give the specifications for this algorithm. As before we use capital letters to indicate variables that can assume multiple-precision values and small letters to indicate single-precision variables.

c ← GIGCDS(a,b)

[Gaussian Integer Greatest Common Divisor, Single-Precision. a and b are Gaussian integers with $|rp(a)|, |ip(a)|, |rp(b)|, |ip(b)| < \beta/4$ where β is the largest single-precision integer of the form 2^{ζ}, ζ a positive integer. c = gcd(a,b).]

Now we present the new algorithm.

C ← GIGCD(A,B)

[Gaussian Integer Greatest Common Divisor. A and B are Gaussian integers. C is set to gcd(A,B).]

(1) [A = 0 or B = 0?] If A = 0 then (C←fqa(B); return); if B = 0 then (C←fqa(A); return).
(2) [Initalize.] $k \leftarrow \zeta - 2$.
(3) [Multiple-precision division.] C ← rem(A,B); A ← B; B ← C; if B = 0 then (C←fqa(A); return).
(4) [Compute lengths.] $\ell \leftarrow L_2^*(A)$; $m \leftarrow L_2^*(B)$; $n \leftarrow \max(\ell,m)$; $h \leftarrow n-k$; if $h \le 0$ then goto (10); if $m - h \le 2$ then goto (3).
(5) [Truncate.] $a_1 \leftarrow [A/2^h]$; $a_2 \leftarrow [B/2^h]$.
(6) [Initialize for single-precision loop.] $u_1 \leftarrow 1$; $u_2 \leftarrow 0$; $v_1 \leftarrow 0$; $v_2 \leftarrow 1$; $i \leftarrow 1$.
(7) [Single-precision divide.] $q \leftarrow \{a_1/a_2\}$; $a_3 \leftarrow rem(a_1,a_2)$; $u_3 \leftarrow u_1 - qu_2$; $v_3 \leftarrow v_1 - qv_2$.
(8) [Termination test for single-precision loop.] $t \leftarrow \max(|rp(a_3)|, |ip(a_3)|)$; $w \leftarrow |rp(u_3)| + |ip(u_3)| + |rp(v_3)| + |ip(v_3)|$; if $t/6 < w$ then goto (9); $a_1 \leftarrow a_2$; $a_2 \leftarrow a_3$; $u_1 \leftarrow u_2$; $u_2 \leftarrow u_3$; $v_1 \leftarrow v_2$; $v_2 \leftarrow v_3$; $i \leftarrow i+1$; goto (7).
(9) [Compute linear combinations of single-precision cosequences and multiple-precision remainders.] If i = 1 then goto (3); $C^* \leftarrow u_1A + v_1B$; $D^* \leftarrow u_2A + v_2B$; $A \leftarrow C^*$; $B \leftarrow D^*$; goto (4).
(10) [Use single-precision gcd algorithm.] C ← GIGCDS(A,B); return. ■

Before proceeding to verify that GIGCD is correct, a few comments are in order. Steps (1) and (2) just check for obvious special cases and initialize k. A multiple-precision division is carried-out initially in step (3) to insure that $|A| \ge |B|$. $|A| \ge |B|$ insures that $|a_1| \ge |a_2|$ in step (5) and that the initial part of a valid remainder sequence for a_1 and a_2 is computed by the loop in steps (7) and (8). A further reason for carrying out a multiple-precision division initially is that a multiple-precision division must be carried out when $|A/B|$ is large. This happens rarely when $A = a_i$ and $B = a_{i+1}$, $i > 1$, are members of a remainder sequence (this is the fundamental reason the algorithm is fast), but it is most likely to happen for i = 1, that is with the initial inputs. So by always performing step (3) initally, a single pass through steps (4) - (9) is avoided in those cases in which the first quotient is large.

In step (4) if $h \le 0$ then we know that the magnitudes of the real and imaginary parts of both A and B are less than $\beta/4$ and the single-precision algorithm GIGCDS can be used.

In step (4) if $m - h < 2$ then $n - m \ge k - 1$. Since n - m is an approximation to the number of bits in $L_2^*(\{A/B\})$, $n - m \ge k - 1$ indicates that $\{A/B\}$ may be multiple-precision and hence requiring a multiple-precision division algorithm to compute it. Otherwise $m - h \ge 2$ and $h \ge 1$ implies that $|B| \ge 4$, and this fact is used in the proof of Theorem 6 which forms a portion of the validity proof for the algorithm.

To demonstrate that the algorithm works correctly we must show that:

(i) The single-precision variables in steps (5) - (8) are assigned only values whose real and imaginary parts are single-precision, i.e., $\le \beta$.

(ii) The algorithm terminates.

In Theorem 4 we show that qu_2 and qv_2 can be computed in step (7) using single-precision arithmetic. Theorem 5 shows that the u_i, v_i and w in steps (7) and (8) always have single-precision real and imaginary parts. Then in Theorem 6 we show that the algorithm terminates.

Steps (2), (4) and (5) guarantee that

$$|rp(a_1)|, |ip(a_1)|, |rp(a_2)|, |ip(a_2)| < \beta/4 \quad (30)$$

each time the single-precision loop is initialized. Since all values assumed for a_1 and a_2 in the single-precision loop are values in a remainder sequence for the initial values of a_1 and a_2, it follows from (26) that the values $|rp(a_1)|$, $|ip(a_1)|$, $|rp(a_2)|$ and $|ip(a_2)|$ remain bounded by $\beta/4$.

Theorem 4. In step (7) of GIGCD, $|qu_2| < \beta/4$ and $|qv_2| < \beta/4$.

Proof. The proof is long and is postponed to the appendix. ∎

Theorem 5. Suppose $\beta > 4$. Then for $1 \le i \le 3$, $|rp(u_i)|$, $|ip(u_i)|$, $|rp(v_i)|$, $|ip(v_i)| < \beta/4$ and $w < \beta$ in steps (7) and (8) of GIGCD.

Proof. Since $w = |rp(u_3)| + |ip(u_3)| + |rp(v_3)| + |ip(v_3)|$, $w < \beta$ follows immediately from the first part of the theorem which we now prove for $i = 3$. The cases for $i = 1$ and $i = 2$ follow in a similar manner.

Let $a_1, a_2, a_3, \ldots, a_{r+1}$ be the least-remainder sequence for a_1, a_2 calculated in step (5) and let $q_1, \ldots, q_{r-1}$; $u_1^*, \ldots, u_{r+1}^*$; and $v_1^*, \ldots, v_{r+1}^*$ denote the associated quotient sequence, the first cosequence, and the second cosequence respectively. Then the values assumed by u_3 and v_3 in step (7) are among the values u_{i+1}^* and v_{i+1}^*, $1 \le i \le r$. But the values assumed by qu_2 and qv_2 in step (5) are among the values $q_i u_{i+1}^*$ and $q_i v_{i+1}^*$. But by Theorem 4 $|q_i u_{i+1}^*| < \beta/4$ and $|q_i v_{i+1}^*| < \beta/4$ which implies that $|u_{i+1}^*|$, $|v_{i+1}^*| < \beta/4$ for $i < r$ since $|q_i| \ge 1$. For $i = r$ we have from (29) that

$$u_{r+1}^* a_1 + v_{r+1}^* a_2 = 0. \qquad (31)$$

But since $\gcd(u_i, v_i) = 1$ for $1 \le i \le r+1$ (this is easily proved by induction), (31) implies that $u_{r+1}^* | a_2$ and $v_{r+1}^* | a_1$. Thus $|u_{r+1}^*| \le |a_2| < \beta/4$ and $|v_{r+1}^*| \le |a_1| < \beta/4$. ∎

Unlike the Lehmer algorithm the q's computed in step (7) may not be members of the quotient sequence for A and B. But the following theorem guarantees that for C* and D* computed in step (9), $|C^*| < |A|$ and $|D^*| < |B|$. Hence the algorithm must terminate.

Theorem 6. Let $A = a2^h + \bar{a}$ with $|rp(\bar{a})|$, $|ip(\bar{a})| < 2^h$. Let $B = b2^h + \bar{b}$ with $|rp(\bar{b})|$, $|ip(\bar{b})| < 2^h$. Let $a_1, \ldots, a_r$ be the least-remainder sequence for a and b, and let $u_1, \ldots, u_r$ and $v_1, \ldots, v_r$ be the corresponding cosequences. Let $A_i = u_i A + v_i B$. If for $i \ge 2$ $|u_i| + |v_i| \le (1/6)|a_i|$ and $|u_{i+1}| + |v_{i+1}| \le (1/6)|a_{i+1}|$ then $|A_{i+1}| < |A_i|$.

Proof. $A_i = u_i A + v_i B = u_i(a2^h + \bar{a}) + v_i(b2^h + \bar{b}) = a_i 2^h + u_i\bar{a} + v_i\bar{b}$, so $|A_i| \ge |a_i|2^h - |u_i|2^h - |v_i|2^h \ge 5/6|a_i|2^h$. Similarly $|A_{i+1}| \le |a_{i+1}|2^h + |u_{i+1}|2^h + |v_{i+1}|2^h \le (7/6)|a_{i+1}|2^h$. But $|a_{i+1}| \le (1/\sqrt{2})|a_i|$ since $i \ge 2$, so $|A_{i+1}| \le (7/6\sqrt{2})|a_i|2^h \le (6/5)(7/6\sqrt{2})|A_i| = (7/5\sqrt{2})|A_i| < |A_i|$. ∎

Note that if $6(|rp(u_i)| + |ip(u_i)| + |rp(v_i)| + |ip(v_i)| \le \max(|rp(a_i)|, |ip(a_i)|)$, then $|u_i| + |v_i| \le (1/6)a_i$, and this is the test that is used to step (8) to determine when to terminate the single-precision loop.

Now we present some empirical data on the performance of GIGCD. Three algorithms were programmed in SAC-1 and their computing times on a Univac 1110 computer were observed. Each of the following tables gives "computing times" in millions of memory references. (The Univac 1110 does not have a CPU clock although it does have a memory reference counter. Memory references are hence used as a measure of CPU time.)

In each table the algorithms are applied to inputs A* = AC and B* = BC with $A = A_1 + A_2 i$, $B = B_1 + B_2 i$, $C = C_1 + C_2 i$. A_i, B_i, C_i are randomly generated integers with $L_\beta(A_1) = L_\beta(A_2) = L_\beta(B_1) = L_\beta(B_2) = n - k$ and $L_\beta(C_1) = L_\beta(C_2) = k$, $1 \le n \le 10$, $0 \le k < n$ (when k=0,C=1). $\beta = 2^{33}$, so each input is a product of two Gaussian integers with real and imaginary parts n - k and k machine words long where each word has 33 bits.

Additional empirical studies show that step (7) is executed approximately 8.5 times each time step(6) is executed. That is, approximately 7.5 values for q are generated each time the single-precision loop in steps (7) and (8) is executed.

n \ k	0	1	2	3	4	5	6	7	8	9
1	.06									
2	.20	.20								
3	.37	.37	.21							
4	.54	.51	.40	.25						
5	.74	.70	.58	.45	.28					
6	.97	.93	.82	.66	.48	.30				
7	1.17	1.16	1.01	.85	.72	.53	.30			
8	1.42	1.38	1.25	1.11	.92	.79	.58	.35		
9	1.72	1.65	1.50	1.33	1.20	1.00	.82	.62	.35	
10	1.96	1.93	1.83	1.68	1.47	1.29	1.12	.89	.66	.39

Table 1.
Univac 1110 Computing Times for GIGCD in Millions of Memory References.

The next table presents computing times for a Lehmer-type algorithm using exact complex interval arithmetic.

n \ k	0	1	2	3	4	5	6	7	8	9
1	.16									
2	.45	.36								
3	.85	.70	.30							
4	1.24	.93	.88	.48						
5	1.56	1.39	1.24	.93	.50					
6	2.15	2.25	2.07	1.22	1.30	.52				
7	3.14	2.63	2.07	2.23	1.55	1.09	.60			
8	3.53	3.34	2.67	2.75	1.86	1.78	1.21	.77		
9	3.99	4.37	3.56	3.08	2.88	2.62	2.31	1.29	.80	
10	4.33	5.27	4.69	3.73	4.18	3.03	2.66	2.05	1.49	.68

Table 2. Univac 1110 Computing Times for a Lehmer-Type Algorithm. Computing Times are Expressed in Millions of Memory References.

The next table gives computing times for the classical Euclidean algorithm using GINQR for the divisions. The time for the Euclidean algorithm is approximately doubled when division is based on (3).

n \ k	0	1	2	3	4	5	6	7	8	9
1	.26									
2	1.00	.65								
3	1.82	1.51	.58							
4	2.64	2.18	1.63	.93						
5	3.62	2.96	2.62	1.90	.98					
6	4.58	4.24	3.51	2.97	2.13	1.10				
7	5.99	5.49	4.11	4.23	3.13	2.22	.98			
8	7.22	7.00	5.73	5.42	4.28	3.52	2.72	1.55		
9	8.25	8.20	7.77	6.26	5.95	4.90	3.99	3.02	1.39	
10	9.60	10.4	9.43	8.50	7.67	6.61	5.50	4.53	3.29	*

Table 3. Univac 1110 Computing Times for the Euclidean Algorithm Using GINQR for Division. Computing Times are Expressed in Millions of Memory References.

We end with a table that shows that GIGCD is up to 5.39 times as fast as the Euclidean algorithm using GINQR for division.

n \ k	0	1	2	3	4	5	6	7	8	9
1	4.30									
2	5.00	3.25								
3	4.92	4.08	2.76							
4	4.89	4.28	4.07	3.72						
5	4.89	4.23	4.52	4.24	3.50					
6	4.72	4.56	4.28	4.50	4.44	3.67				
7	5.12	4.73	4.07	4.98	4.35	4.19	2.80			
8	5.08	5.08	4.58	4.88	4.65	4.46	4.69	4.43		
9	4.80	4.97	5.18	4.71	4.96	4.90	4.87	4.84	3.97	
10	4.90	5.39	5.16	5.06	5.22	5.12	4.91	5.09	4.98	*

Table 4. Ratio of Euclidean Algorithm Times to GIGCD Times

Appendix - Proof of Theorem 4.

The proof of Theorem 4 requires a number of supporting results which we present first. Let $a = a_1$, $b = a_2,\ldots,a_{r+1}$ be the least-remainder sequence of the Gaussian integers a and b with quotient sequence $q_1,\ldots,q_{r-1}$ and first and second cosequences $u_1,\ldots,u_{r+1}$ and $v_1,\ldots,v_{r+1}$.

Lemma 7. $-\frac{1}{2} < rp(\frac{a_{i+2}}{a_{i+1}}),\ ip(\frac{a_{i+2}}{a_{i+1}}) \le \frac{1}{2}$ and

$\left|\frac{a_{i+2}}{a_{i+1}}\right| \le 1/\sqrt{2}$ for $1 \le i < r$.

Proof. We prove that $\{a_{i+1}/a_i\} = 0$ for $1 < i \le r$ from which the desired result follows immediately. $a_i = q_i a_{i+1} + a_{i+2}$ for $1 \le i < r$. Thus $q_i = \{a_i/a_{i+1}\} = \{q_i + a_{i+2}/a_{i+1}\} = q_i + \{a_{i+2}/a_{i+1}\}$ since q_i is a Gaussian integer and hence $\{a_{i+2}/a_{i+1}\} = 0$ for $1 \le i < r$. ∎

For any complex number z, let S_z be the square $\{u:\{u\} = z\}$ and let C_z be the circle $\{u:|u-z| = 1\}$.

Lemma 8. $|q_i| \ge \sqrt{2}$ for $2 \le i \le r-1$.

Proof. Let $1 \le i < r-1$. By Lemma 7, $\frac{a_{i+2}}{a_{i+1}} \in S_0$. Let $\emptyset$ be the mapping $\emptyset(z) = z^{-1}$. It is well known (see, e.g., Analytic Functions of a Complex Variable, by D. R. Curtiss, Section 29) that $\emptyset$ map circles onto circles, where a straight line is regarded as a circle with infinite radius. Since $(\frac{1}{2}+\frac{1}{2}i)^{-1} = 1-i$, $(\frac{1}{2}-\frac{1}{2}i)^{-1} = 1+i$ and $(\frac{1}{2})^{-1} = 2$, we see that $\emptyset$ maps the line $rp(z) = \frac{1}{2}$ onto the circle C_1. Similarly, the lines $rp(z) = -\frac{1}{2}$, $ip(z) = \frac{1}{2}$ and $ip(z) = \frac{1}{2}$ are mapped onto C_{-1}, C_{-i} and C_i, respectively. See Figure 2. It follows that $\frac{a_{i+1}}{a_{i+2}} = (\frac{a_{i+2}}{a_{i+1}})^{-1}$ is on or exterior to each of these four circles. Since S_0, S_1, S_i, S_{-1} and S_{-i} are all contained in the union of the interiors of these circles it follows that $q_{i+1} = \left\{\frac{a_{i+1}}{a_{i+2}}\right\}$ is none of the points 0, 1, i, -1 or -i. Thus $|q_{i+1}| \ge \sqrt{2}$. ∎

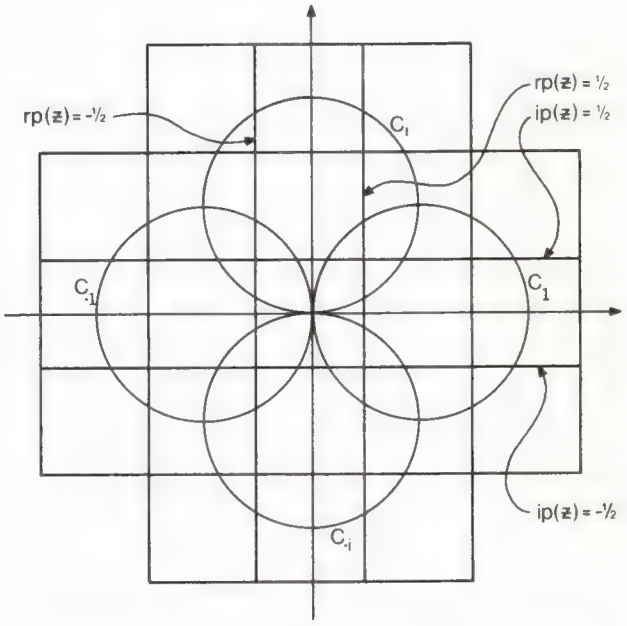


Figure 2.

Lemma 9. $u_i \ne 0$ for $3 \le i \le r$. $v_i \ne 0$ for $2 \le i \le r+1$.

Proof. $v_2 = 1$ so assume $3 \le i \le r$. Then $u_i a + v_i b = a_i$. If $u_i = 0$ then $v_i b = a_i \ne 0$. Hence $b|a_i$ and $|b| \le |a_i|$. But $|a_i| < |b|$ by

Lemma 8, a contradiction. Hence $u_i \neq 0$, and similarly $v_i \neq 0$. ∎

Let $\bar{u}_i = -u_{i+2}/u_{i+1}$ for $2 \leq i \leq r-1$, $\bar{v}_i = -v_{i+2}/v_{i+1}$ for $1 \leq i \leq r-1$.

Lemma 10. $\bar{u}_2 = q_2$ and $\bar{u}_i = q_i + (\bar{u}_{i-1})^{-1}$ for $3 \leq i \leq r-1$. $\bar{v}_1 = q_1$ and $\bar{v}_i = q_i + (\bar{v}_{i-1})^{-1}$ for $2 \leq i \leq r-1$.

Proof. $u_3 = u_1 - q_1 u_2 = 1 - q_1 \cdot 0 = 1$ and $u_4 = u_2 - q_2 u_3 = 0 - q_2 \cdot 1 = -q_2$, so $\bar{u}_2 = -u_4/u_3 = q_2$. Let $3 \leq i \leq r-1$. Then $\bar{u}_i = -u_{i+2}/u_{i+1} = -(u_i - q_1 u_{i+1})/u_{i+1} = q_i - u_i/u_{i+1} = q_i + (\bar{u}_{i-1})^{-1}$. The proof of the second part is similar. ∎

Lemma 11. If $q_i = 1+i$ and $2 \leq i < r-1$, then either q_{i+1} is in the fourth quadrant or $rp(q_{i+1}) = -1$ and $ip(q_{i+1}) \leq -2$, or $ip(q_{i+1}) = 1$ and $rp(q_{i+2}) \geq 2$. Similar results hold for $q_i = 1-i$, $q_i = -1+i$ and $q_i = -1-i$.

Proof. By the proof of Lemma 8, $\frac{a_i}{a_{i+1}}$ is on or outside C_1 and C_i, while $q_i = \left\{\frac{a_i}{a_{i+1}}\right\} = 1+i$. Hence $\frac{a_{i+2}}{a_{i+1}} = \frac{a_i}{a_{i+1}} - q_i$ is on or outside C_{-i} and C_{-1}, but in S_0. Now $\emptyset$ maps C_{-i} onto the line $rp(z) = -1/2$, C_{-1} onto the line $ip(z) = 1/2$, maps the line $ip(z) = 1/2$ onto C_{-i}, and maps the line $rp(z) = 1/2$ onto C_1. So $\frac{a_{i+1}}{a_{i+2}} = \left(\frac{a_{i+2}}{a_{i+1}}\right)^{-1}$ lies on or outside the circles C_1 and C_{-i}, on or to the right of the line $rp(z) = -1/2$, and on or below the line $ip(z) = 1/2$. Since $q_{i+1} = \left\{\frac{a_{i+1}}{a_{i+2}}\right\}$, it follows that q_{i+1} is in the fourth quadrant (including its boundaries), or $rp(q_{i+1}) = -1$ and $ip(q_{i+1}) \leq -2$, or $ip(q_{i+1}) = 1$ and $rp(q_{i+1}) \geq 2$. ∎

Lemma 11 can be slightly strengthened, e.g., if $q_i = 1+i$ then q_{i+1} must be in the fourth quadrant (including the boundary $ip(q_{i+1})=0$) but the strengthened form is not needed here.

Lemma 12. Let $3 \leq i \leq r-1$. If $|q_i| \neq \sqrt{2}$ then $|\bar{u}_i| \geq 1$. If $|q_i| = \sqrt{2}$ and $|q_{i-1}| \neq \sqrt{2}$, then $|q_i - \bar{u}_i| \leq 1$. If $|q_i| = \sqrt{2}$ and $|q_{i-1}| = \sqrt{2}$, then $|\bar{u}_i| \geq 1$ and $\bar{u}_i$ is in the same quadrant as q_i.

Proof. By induction on i. For $i = 3$, $\bar{u}_2 = q_2$ so $\bar{u}_3 = q_3 + q_2^{-1}$. If $|q_3| \neq \sqrt{2}$ then $|\bar{u}_3| = |q_3 + q_2^{-1}| \geq |q_3| - |q_2^{-1}| \geq 2 - 1/\sqrt{2} > 1$. If $|q_3| = \sqrt{2}$ and $|q_2| \neq \sqrt{2}$ then $|q_3 - \bar{u}_3| = |q_2^{-1}| \leq \frac{1}{2}$. If $|q_3| = \sqrt{2}$ and $|q_2| = \sqrt{2}$ then by Lemma 11 $q_3 = \bar{q}_2$, the conjugate of q_2. Hence $q_2^{-1} = \frac{1}{2}\bar{q}_2 = \frac{1}{2}q_3$, $|\bar{u}_3| = |q_3 + \frac{1}{2}q_3| = \frac{3}{2}|q_3| = 3/\sqrt{2} > 1$, and $\bar{u}_3 = \frac{3}{2}q_3$ is in the same quadrant as q_3.

Assume $3 < i < r-1$. If $|q_{i+1}| \neq \sqrt{2}$ and $|q_i| \neq \sqrt{2}$ then $|q_{i+1}| \geq 2$ and $|\bar{u}_i| \geq 1$ by the induction hypothesis, so $|\bar{u}_{i+1}| = |q_{i+1} + \bar{u}_i^{-1}| \geq |q_{i+1}| - |\bar{u}_i^{-1}| \geq 2 - 1 = 1$.

Assume $|q_{i+1}| \neq \sqrt{2}$ and $|q_i| = \sqrt{2}$. If $|q_{i-1}| \neq \sqrt{2}$ then $|q_i - \bar{u}_i| \leq 1$ by the induction hypothesis. Hence q_i and $\bar{u}_i$ are in the same quadrant. Hence $\bar{u}_i^{-1}$ is in the quadrant conjugate to that of q_i. By Lemma 11, either q_{i+1} is in the quadrant conjugate to that of q_i, in which case $|\bar{u}_{i+1}| = |q_{i+1} + \bar{u}_i^{-1}| \geq |q_{i+1}| \geq 2$, or else (in case $q_i = 1+i$) $rp(q_{i+1}) = -1$ and $ip(q_{i+1}) \leq -2$, in which case $|\bar{u}_{i+1}| = |q_{i+1} + \bar{u}_i^{-1}| \geq |ip(q_{i+1})| \geq 2$, or else $ip(q_{i+1}) = 1$ and $rp(q_{i+1}) \geq 2$, in which case $|\bar{u}_{i+1}| \geq |rp(q_{i+1})| \geq 2$. If $|q_{i-1}| = \sqrt{2}$ then by the induction hypothesis $|\bar{u}_i| \geq 1$, so $|\bar{u}_{i+1}| = |q_{i+1} + \bar{u}_i^{-1}| \geq |q_{i+1}| - |\bar{u}_i^{-1}| \geq 2 - 1 = 1$.

Assume $|q_{i+1}| = \sqrt{2}$ and $|q_i| \neq \sqrt{2}$. Then $|\bar{u}_i| \geq 1$ by the induction hypothesis. Hence $|\bar{u}_{i+1} - q_{i+1}| = |\bar{u}_i^{-1}| \leq 1$.

Assume $|q_{i+1}| = \sqrt{2}$ and $|q_i| = \sqrt{2}$. By Lemma 11, $q_{i+1} = \bar{q}_i$. If $|q_{i-1}| \neq \sqrt{2}$ then by induction hypothesis $|q_i - \bar{u}_i| \leq 1$. Hence $\bar{u}_i$ is in the same quadrant as q_i and $\bar{u}_i^{-1}$ is in the same quadrant as q_{i+1}. Hence $\bar{u}_{i+1} = q_{i+1} + \bar{u}_i^{-1}$ is in the same quadrant as q_{i+1} and $|\bar{u}_{i+1}| = |q_{i+1} + \bar{u}_i^{-1}| \geq |q_{i+1}| = \sqrt{2}$. If $|q_{i-1}| = \sqrt{2}$ then by the induction hypothesis $|\bar{u}_i| \geq 1$ and $\bar{u}_i$ is in the same quadrant as q_i. Again $\bar{u}_i^{-1}$ is in the same quadrant as q_{i+1}, $\bar{u}_{i+1} = q_{i+1} + \bar{u}_i^{-1}$ is in the same quadrant as q_{i+1}, and $|\bar{u}_{i+1}| \geq |q_{i+1}| = \sqrt{2}$. ∎

By induction one obtains from (27) and (28) that

$$u_i a_{i+1} - u_{i+1} a_i = (-1)^{i+1} b \tag{32}$$

$$v_i a_{i+1} - v_{i+1} a_i = (-1)^{i+1} a \ . \tag{33}$$

(32) and (33) are used in the following lemma.

Lemma 13. If $2 \le i \le r-1$ and $|\bar{u}_i| \ge 1$, then $|u_{i+2}| \le (2+\sqrt{2}) \cdot |b/a_{i+1}|$.

Proof. By (32)

$$b = (-1)^{i+1} a_{i+1} u_{i+2} + (-1)^{i} a_{i+2} u_{i+1} . \tag{34}$$

Dividing (34) by a_{i+1} and rearranging, we obtain

$$u_{i+2} \left(1 - \frac{a_{i+2}}{a_{i+1}} \cdot \frac{u_{i+1}}{u_{i+2}} \right) = (-1)^{i+1} b/a_{i+1} \ .$$

Hence, using the definition of $\bar{u}_i$, we have that

$$|u_{i+2}| \cdot \left| 1 - \frac{a_{i+2}}{a_{i+1}} \cdot \bar{u}_i^{-1} \right| = |b/a_{i+1}| .$$

By Lemma 7, $\left| \frac{a_{i+2}}{a_{i+1}} \right| \le 1/\sqrt{2}$, and by hypothesis, $|\bar{u}_i^{-1}| \le 1$. Hence

$$|u_{i+2}| \cdot (1-1/\sqrt{2}) \le |b/a_{i+1}| .$$

But $(1-1/\sqrt{2})^{-1} = 2+\sqrt{2}$. ∎

Lemma 14. $|u_i| \le (2+\sqrt{2}) \cdot |b/a_{i-1}|$ (35)

for $2 \le i \le r+1$.

Proof. (35) holds trivially for $i = 2$ and $i = 3$ since $u_2 = 0$ and $u_3 = 1$. It holds for $4 \le i \le r+1$ by Lemma 13 whenever $|\bar{u}_{i-2}| \ge 1$. So assume $4 \le i \le n+2$ and $|\bar{u}_{i-2}| \le 1$. Then $i \ge 5$ since $|\bar{u}_2| = |q_2| \ge 1$. Hence by Lemma 11, $|q_{i-2}| = \sqrt{2}$ and $|q_{i-3}| \ne \sqrt{2}$. If $i = 5$ then $|\bar{u}_{i-3}| = |q_2| \ge 1$ and otherwise by Lemma 11, $|\bar{u}_{i-3}| \ge 1$. Hence by Lemma 13,

$$|u_{i-1}| \le (2+\sqrt{2}) \cdot |b/a_{i-2}| . \tag{36}$$

Since $\frac{a_{i-2}}{a_{i-1}} \le 1/\sqrt{2}$, (36) implies

$$|u_{i-1}| \le (1+\sqrt{2}) \cdot |b/a_{i-1}| .$$

Also, $|u_i/u_{i-1}| = |\bar{u}_{i-2}| < 1$, so $|u_i| < |u_{i-1}|$ and hence $|u_i| \le (1+\sqrt{2}) \cdot |b/a_{i-1}|$. ∎

Lemma 15. For $2 \le i \le r+1$,

$$|v_i| \le (3+\sqrt{2}) \cdot |a/a_{i-1}| .$$

Proof. By (29), $u_i a + v_i b = a_i$. Hence

$$|v_i b| \le |u_i a| + |a_i| . \tag{37}$$

By (37) and Lemma 14,

$$|v_i b| \le (2+\sqrt{2}) \cdot |ab/a_{i-1}| + |a_i| ,$$

and hence

$$|v_i| \le (2+\sqrt{2}) \cdot |a/a_{i-1}| + |a_i/b| .$$

But $|a_i/b| \le 1 \le |a/a_{i-1}|$ so $|v_i| \le (3+\sqrt{2})|a/a_{i-1}|$. ∎

Now we restate and prove Theorem 4.

Theorem 4. In step (7) of GIGCD, $|qu_2| < \beta/4$ and $|qv_2| < \beta/4$.

Proof. Let $a_1, a_2, \ldots, a_{r+1}$ be the least-remainder sequence for a_1 and a_2 calculated in step (5) with associated quotient sequence $q_1, \ldots, q_{r-1}$ and cosequences $u_1^*, \ldots, u_{r+1}^*$ and $v_1^*, \ldots, v_{r+1}^*$. For some $i \ge 1$, the value of q is q_i and the values of u_2 and v_2 are u_{i+1}^* and v_{i+1}^*. Since $q_i = \{a_i/a_{i+1}\}$, we have $|q_i| \le |a_i/a_{i+1}| + 1 \le 2|a_i/a_{i+1}|$. For $i = 1$, $u_{i+1}^* = 0$ so $|q_i u_{i+1}^*| = 0 < \beta/2$ and $v_{i+1}^* = 1$ so $|q_i v_{i+1}^*| = |q_i| \le 2|a_1/a_2| \le \frac{1}{2}|a_1| < |\sqrt{2}/2| \cdot |a_1| \le 2^k = \beta/4$ since $|a_2| \ge 4$ by the test "if m-h < 2" in step (4).

Now assume $i \ge 2$. By the termination test in step (8), $|a_{i+1}| \ge 6|u_{i+1}^*| + |v_{i+1}^*| \ge 6(1+\sqrt{2}) \ge 14$. By Lemma 14, $|u_{i+1}^*| \le (2+\sqrt{2})|a_2/a_i|$ so $|q_i u_{i+1}^*| \le 2|a_i/a_{i+1}| \cdot (2+\sqrt{2})\,|a_2/a_i| = (4+2\sqrt{2})|a_2/a_{i+1}| \le ((4+2\sqrt{2})/14)|a_2| \le ((4\sqrt{2}+4)/14)2^k < 2^k = \beta/4$. By Lemma 15, $|v_{i+1}^*| \le (3+\sqrt{2})|a_1/a_i|$ so $|q_i v_{i+1}^*| \le (6+2\sqrt{2})|a_1/a_{i+1}| \le ((4+6\sqrt{2})/14)2^k < 2^k = \beta/4$. ∎

References

[AHU 74] A. V. Aho, J. E. Hopcroft and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley Publishing Co., Reading, Mass., 1974.

[Cav 73] B. F. Caviness, A Lehmer-Type Greatest Common Divisor Algorithm for Gaussian Integers, SIAM Review Vol. 15, No. 2, Part 1 (April 1973), 414.

[Col 71] G. E. Collins, The Calculation of Multivariate Polynomial Resultants, J. Assoc. Comput. Mach. Vol. 18, No. 4 (Oct. 1971), 515-532.

[Leh 38] D. H. Lehmer, Euclid's Algorithm for Large Numbers, Amer. Math. Monthly Vol. 45, No. 4 (April 1938), 227-233.

[ScS 71] A Schönhage and V. Strassen, Schnelle Multiplikation Grosser Zahlen, Computing Vol. 7 (1971), 281-292.

A New Reduce Model for Algebraic Simplification*

Anthony C. Hearn
University of Utah

Abstract

This paper shows how the general concepts of mode analysis can play a useful role in the design and implementation of programs for algebraic simplification. We utilize these ideas in presenting a new model for simplification which will be incorporated into REDUCE in the near future.

1. Introduction

This paper shows how the concepts of mode or type analysis can play a useful role in the design and implementation of programs for the simplification of algebraic expressions. The result is more flexible and more easily maintainable algebraic manipulation programs, clearly goals of utmost importance in such a rapidly developing field. Such goals can also be achieved without any loss of efficiency; in fact in essentially all cases more efficient programs result. As a consequence of this study we shall also present a new model for expression simplification which will be incorporated into REDUCE in the near future.

There are two main aspects to the question of algebraic simplification. The first relates to the problem of designing algorithms for the canonical evaluation of as large a class of expressions as possible. This is a question of much theoretical and practical importance, but not one we are considering here. The second relates to the actual design of programs to incorporate such theoretical advances. This is also a problem of great practical importance to users of algebra systems, because if the inclusion of new theoretical results in an existing program requires major reprogramming, then it will be many months (or maybe never!) before such users see the effect of these results on their calculations. The resolution of this problem in REDUCE was in fact one of the reasons this study was begun. In particular, the desirability of utilizing the results of Brown and Hall's (1, 2) study of algorithms for factored forms as a general facility in REDUCE was a major inducement. However, since incorporating this representation into the current system would have required considerable changes to the source program, it was decided to reexamine the whole basis for the model so that when other theoretical improvements come later (as surely they will!) they can be incorporated with the minimum of effort.

Another motivation for this study was the obvious need for a general model of simplification which was more easily explainable to users than the current one. Several recent applications of REDUCE have required this level of explanation (3) and it was therefore decided to attempt a description in terms of the mode concepts which are currently being introduced into REDUCE. This analysis revealed in fact many inefficiencies in the current implementation. Our experience in this regard is probably little different from other designers of large systems; as users have discovered special cases for which the available program did a poor job of simplifying or required some special facility for their particular problems, local patches were made to take care of this. However these patches sometimes repeated work done by another section of the program so that inefficiencies arose. Several such examples were revealed by our study and resolved in the new model presented here.

Because the ideas of mode analysis may be new to some readers, we shall summarize in Section 2 the salient features necessary for this study. In Section 3 we shall present a description of the new simplification model. We shall then show in Section 4 how the mode ideas provide for a natural implementation of the simplification model and at the same time provide the flexibility necessary to make future maintenance of the program much easier.

2. Mode Analysis

Since the ideas of mode analysis are important in our study of simplification, we shall repeat in this section a brief description of the process as it is being implemented in a new version of REDUCE (4). These ideas are not new, and the choices made in the implementation have been influenced by a study of the features found in EL1 (5), ALGOL 68 (6) and PASCAL (7). We associate the term *modes* with the various data-structures considered by REDUCE. Currently the built-in modes include *integer* (i.e., arbitrary precision integers), *real* and *algebraic*. The meaning of the latter mode will be explained later. A mode *literal* is also defined for a literal atom, and an

*Work supported in part by the National Science Foundation under Grant No. GJ-32181.

atom itself (i.e., a literal or a number) is of mode *atom*.

In addition to its value, every literal in REDUCE has such a mode associated with it. This mode may be provided by the user by means of a DECLARE statement or defaults to a standard type otherwise (usually algebraic).

Each expression built up from the basic literals and constants also has a mode associated with it, determined by a semantic analysis of the expression. This analysis, performed by a function MEVAL, is just one step in the complete evaluation of an expression. Each command is in fact processed in four separate stages; a parse, a mode analysis, an (optional) compilation and an evaluation. However, the mode analyzer is also available (as a REDUCE procedure) for use during evaluation, and this use will prove important to our discussion here.

In the parse step, each command is converted into an equivalent LISP expression using standard LISP names for the infix arithmetic operators. For example, the REDUCE expression (X+A) * (Y+B) becomes (TIMES (PLUS X A) (PLUS Y B)). Following a suggestion of M. Nordstrom, this conversion attempts to preserve as closely as possible the structure of the input expression, and is therefore a fairly faithful representation of the parse tree for the expression. In the current version of REDUCE, however, the parser also applies some semantic transformations to the data. For example, FOR statements are converted into blocks. This complicates the parser by introducing material more concerned with semantics than syntax and makes compilation harder by throwing away control information too early. The parser in the new version is essentially context and semantic free and is based on one described and implemented by Nordstrom (8).

The design of the mode evaluator is similar to that of any LISP evaluator. It takes an expression as argument, modifies it according to the semantic analysis and returns a dotted pair of the modified expression and its mode. The analysis proceeds roughly as follows. If the argument is an atom, it is either a constant (for example, a number) or a literal; in either case its mode is known. Otherwise the expression is assumed to be of the form f(arg1,...,argn), or (f arg1 ... argn) in LISP notation. A check is first made to see if f is a special form as far as the mode evaluator is concerned. If it is, a specific mode analyzing function is used. In the usual case, however, MEVAL will begin by performing a mode analysis of the arguments of the function, determining in the process their mode. It will then attempt to assign a meaning to f with arguments of this mode by an inspection of a table on the property list of f. If a meaning can be assigned, the appropriate internal function to use in that case will be determined, otherwise an error results. This process may involve coercion (or conversion), in other words a transformation from one mode to another. A simple numerical example illustrates this. Consider the program segment

```
DECLARE A,B:INTEGER;  A+B;
```

The second statement will be parsed as (PLUS A B). An inspection of the semantic properties of the generic function PLUS will reveal that the addition of two integers has meaning, and that the internal function associated with this process is, say, IPLUS whose value has the mode *integer*. MEVAL will therefore return the expression ((IPLUS A B) . INTEGER). However, suppose instead that we had said:

```
DECLARE A:INTEGER,B:REAL;  A+B;
```

The parse of the last statement will still be the same, but the mode analyzer will now find that no routine exists for the addition of an integer and a real number. It will find however, that it is possible to convert an integer into a real number and that addition of two real numbers using, say, RPLUS is possible. If the function for doing this conversion is FLOATF, then MEVAL will return in this case ((RPLUS (FLOATF A) B) . REAL). Further details on the use of this facility are given in Ref. (4) and (9).

With these ideas in mind we can now turn to the particular problem under consideration, namely the design and implementation of a simplification model for algebraic expressions.

3. The Simplification Model

One word guaranteed to produce sharp reactions and controversy among those concerned with symbolic computation is *simplification* and so it will be necessary to attach our own meaning to the term. We agree with Moses (10) and Brown (1) that simplification changes only the form or representation of an expression. However, we also require that the transformed expression be *idempotent*, namely that if the simplification algorithm is immediately applied again to the simplified expression its form will not change. We shall call this idempotent expression the *value* of the original expression, and to avoid further controversy on the meaning of simplification, shall call the transformation to this value the *evaluation* of the expression. Because an idempotent expression is semantically different from the original class of expressions we began with, it is reasonable to associate a new mode *idempotent* with this class. So the value of an expression of idempotent mode is itself. The simplification algorithm exploits this fact to avoid any further computation when it meets an expression so tagged. (This is reminiscent of the"already simplified" flag of FORMAC (11).) In the current version of REDUCE, such idempotency is not recognized consistently, so that some such expressions are reevaluated by the full algorithm. Although such reevaluation is usually on small expressions so that the additional cost is not high, it is an overhead nonetheless which is eliminated in the new model.

It is of course necessary to recognize that the idempotency of such expressions may be destroyed later by some operation, such as the assignment of a value to one of the literals in an expression. However, such a possibility is easily handled. In the worst case one could change the mode of all such expressions whenever a new sub-

stitution is defined. This naturally is too inefficient to consider in practice and so one tries to keep the class of expressions whose mode is changed as small as possible by keeping track of the literals and operators on which the idempotent expressions depend.

Another important question to resolve in designing an algebraic simplification program is the appropriate interface between the user and the system. In other words, should the system try to leave an expression as close as possible to the form in which the user entered it or immediately transform it into an internal representation which can be manipulated with greater ease by the program? Our current feeling is that if algebra systems wish to attract casual users who do not care to learn the intricacies of the internal representations, then one must choose a programming interface which is as near as possible to the user's input, and an easy choice is the parsed form of the user's expressions.

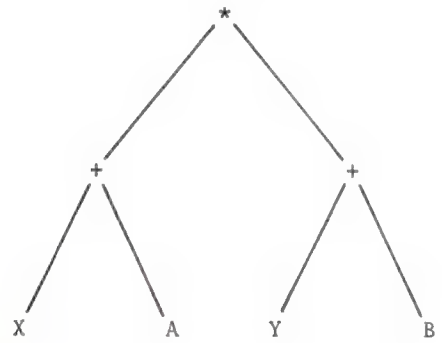


Figure 1. Parse tree for expression (X+A) * (Y+B)

It has proved quite easy to explain to users that the expression (X+A) * (Y+B) is parsed by the program into the form equivalent to Fig. 1, (more precisely, the LISP form given earlier), and therefore concepts like leading operator, *, leading term, (X+A), and so on are quickly apparent. On the other hand, trying to explain to a casual user that the leading term of (X+A) * (Y+B) is A * B, which might be true if an internal representation for polynomials in expanded form is used, even if the expression is displayed as above, can only lead to confusion and frustration. This is not to say that a more expert user should not be allowed direct access to such an internal form; on the contrary. However, such a user should be forced to do something, such as make some additional declarations, in order to get that access.

In a sense, we are reinforcing the point also made by Moses (12) that users like to see their output in terms of the same functions as their input. Moses was talking about integration, but the point is clearly more general. In this spirit we believe that users also like to think about manipulating expressions in terms of the structures they see before them on a terminal or printed output, not in terms of some internal representation possibly bearing little relationship to the output.

It is with this parsed form for expressions that we associate the mode *algebraic*. To keep expressions in the most "natural" form from the user's point of view, it is reasonable to consider expression evaluation by means of pattern matching on the parsed expressions. For this purpose the model provides a general purpose pattern matching facility which transforms such expressions by a Markov algorithm similar to that used by SCRATCHPAD (13). In our model, a *pattern* considered by this facility consists of three parts; a *template*, which is an expression to be matched against, a *boolean expression* defining conditions on free parameters in the template, and a *value* for the original expression if the template matches and the boolean expression is true. Such patterns are introduced by the LET command in REDUCE. Its syntax is

```
<optional boolean expression>
        LET <expression> = <expression>,...
```

Arbitrary (free) parameters in the boolean expression are simply tagged as such and not included in the pattern boolean expression. Otherwise the optional boolean expression becomes the boolean expression in the pattern (which will be *true* if no boolean expression is present in the LET statement), the left hand side of the substitution is the template, and the right hand side the value. For example, the statement

```
LET COS(0) = 1
```

would result in the pattern (COS(0),TRUE,1).

Each operator and literal has associated with it a *pattern list* to which new patterns are added when defined. This list is organized as a stack in that the last pattern is entered at the head of the list. Since this list is searched from the top when pattern matching is attempted the latest pattern entered takes precedence over any previous pattern since the pattern matcher stops as soon as it has a successful match. When a new pattern is added to a pattern list, the system checks to see if the same template and boolean condition already appear on the list, in which case the old pattern is deleted. However, it is important to note that no other check is made for the compatibility of a new pattern with any previously declared patterns, and so a user is not protected against declaring inconsistent patterns by mistake. However, at least the last entry is seen first!

Evaluation of an algebraic expression now proceeds as follows. Such an expression is either an atom or a structure of the form f(arg1,...,argn). If the expression is an atom, it is either a number or a literal. In the former case, a flag is checked to see what global number mode is active, i.e. floating or fixed, and an appropriate conversion takes place if the number is in the wrong mode. In the latter case, the literal pattern list is checked for a true boolean expression and if one is found the algorithm is reapplied to the pattern value until an idempotent expression results. Otherwise the literal is returned unchanged.

On the other hand, if the expression has structure, we begin by evaluating the arguments recursively by the process we are describing. The operator is then checked for any patterns associ-

ated with it. If any are found, a check is made to see if any match the expression. If so, the algorithm is reapplied to the pattern value with any free parameters of course replaced by their relevant values; otherwise the form f(arg1*,...,argn*) is returned, where the asterisks denote the evaluated arguments.

The actual expression matching algorithm is also straightforward. Suppose that we are trying to match the template $f(u_1,...,u_m)$ against the expression $f(v_1,...,v_n)$. If f is neither commutative nor associative then we require that m = n and then match u_1 against v_1, u_2 against v_2 and so on as follows. If a given u_i happens to be an atom, then if it is tagged as free a match has occurred and the free parameter is paired with the expression v_i in a table. If u_i is not free it must equal v_i for a match to occur. On the other hand, if u_i is not an atom then v_i must also be a structure with the same leading operator and the algorithm is then applied recursively to u_i and v_i. If the whole expression matches in this manner, the appropriate values for the free parameters are substituted into the boolean form, which, if true on evaluation, means that a complete match has occurred.

If f is either commutative or associative (or both!) then the matching process has to take into account all possible pairings of arguments allowed by these properties. It is clear that this process could be quite time consuming in the worst case. In the current REDUCE there are, as a result, restrictions on the way that the associativity property of +, for example, is applied in order to make the matching process more efficient. In fact the matching process is only applied to a right binary grouping of the expression plus the permutations allowed by the commutativity rule. We plan to investigate in the new implementation the cost of making the process more general, and if possible remove the restrictions existing in the current system.

If this were the complete model for REDUCE evaluation, then there would be no difficulty in explaining precisely to every user how expressions are evaluated. However, a scheme based solely on pattern matching does not provide for the efficiency inherent in more canonical schemes, or for a natural interface to algorithms based on canonical representations such as factorization. To proved for such internal representations, however, is a fairly simple modification to our model. One associates with a certain class of operators, usually the standard arithmetic operators, +, *, etc. an *internal* property and further assumes that the mode of the value of an expression (i.e. *idempotent*) is a union of two modes; *kernel* and *internal*. The former is an idempotent sub-class of the class of algebraic expressions, whereas the latter can have a representation entirely different. However, we assume the existence of a transformation between internal forms and kernels and vice versa, as well as a run time check for each sub-mode (since *idempotent* is now a union mode). Therefore, if we are dealing with a non-internal operator for which the above pattern matching model still applies, we must first check to see if each argument after the evaluation is of kernel mode and convert it if not. Likewise, the functions for the evaluation of expressions involving top level internal operators must also allow for the existence of the union mode. In most cases, however, the function definition will specify the required mode for its arguments so that the mode evaluator will insert the appropriate conversion function automatically as in our numerical example in Section 2. For simplicity, we also assume that the value of an expression with a top level internal operator is of internal mode.

An obvious choice for such internal representations is to make them canonical in some sense but again to avoid controversy we need not insist on this in our description. However, the standard internal representation currently used in REDUCE (14, 15) is a rational form with numerator and denominator in expanded form, which is canonical if the kernels which act as indeterminates in this representation are independent, and the expression has been reduced to lowest terms.

Rules for substitutions for expressions involving top level canonical operators are also handled quite differently from the previous model. In order to perform the evaluation of such expressions in an optimal manner, the system applies its own set of evaluation rules and those introduced by a user in an arbitrary order. In particular, most user substitutions are only applied when a conversion from *internal* to *kernel* takes place. For this method to work, it is necessary that any top level expression of internal mode be converted to kernel mode in order that an expression be always returned in idempotent form. As a compromise for reasons of efficiency, however, we can often avoid this conversion by applying the substitution to the expression which will return another internal form and tagging this as a "substituted internal form". Of course, if a user wishes to display such an expression and get at parts of it as seen on the display, the full conversion has to be done.

Let me repeat however, that such substitutions for internal operators are applied only once to the whole expression. This has usually proved a more efficient practice than applying the substitution rules every time an internal operator is encountered, and also seems to lead to more compact expressions in some cases. For example, given that I**2 = -1, the expression (I**2-1)/(I+1) will have the value I-1 by this method, whereas it would be -2/(I+1) if the substitution was applied as soon as the exponentiation operator was encountered. An exception to this rule is made for any power of an indeterminate whose value is 0. This is immediately substituted in order to avoid unnecessary computation with zero valued expressions.

For those interested, the order in which such substitutions are applied in the current REDUCE is first rules for powers with *true* boolean expressions and no free parameters, then rules for other powers and finally rules for products. Rules for top level sums and quotients are not allowed in the current system. It is clear that we want to remove this restriction in an efficient manner in the new system. One method being investigated is the possibility of generating an ideal from the set of substitutions and then reducing a given

polynomial or rational function to a unique representative in the appropriate equivalence class (16). Such a method would again apply the rules in an arbitrary order, but since the method checks for the self-consistency of the rules, the results will be independent of their order of application. This method, however, cannot currently incorporate substitutions with free parameters or boolean conditions.

This then is the complete model. We see that canonical evaluation and pattern matching each play a complementary role and that the mode of the value of an expression can either be *kernel* (i.e. an idempotent algebraic expression) or *internal*. This differs from the current REDUCE system (14, 15), where all values are of internal mode, so that a certain amount of overhead is involved in writing kernels as internal expressions and then extracting the kernel again when needed. The new model could also allow for an idempotent value mode which was a union of several different modes corresponding to different internal representations for different operators. Little would need to be changed if this later proves desirable.

We can allow also for users to specify that their rules for internal operators take precedence over the system rules by first using the pattern matching program on such operators before applying the system rules. Such a possibility does in fact exist in the current REDUCE and was used by Stoutemyer (17) to postpone such operations as the collection of like powers and like terms, the distribution of powers over factors and the associative laws of addition and multiplication until after his set of rules had been applied. However, this facility was added in a rather ad hoc manner, whereas in the current model it is an integral part of the algorithm. Another possibility is to allow a user to remove all internal properties on operators and work solely in a pattern matching context for a while so that only his rules are applied.

4. Implementation of the Model

So far, we have described a model for expression evaluation in which the language of mode analysis proved of some help in the description. However, we have not yet faced the problem of implementing this model in a manner which allows for the easy change from one internal form to another; for example to replace an expanded representation for the class of internal operators with say a factored representation. There are in fact three tasks to be considered, namely:

1. writing the routines for arithmetic operations and substitutions on the new structures,

2. writing the routines for transforming between an algebraic expression (our interface) and this new mode,

3. writing the routines for transforming between the internal mode and the user interface and printing expressions in this form.

The first problem is considered in Refs. (4) and (9). In fact Griss (9) shows how the data-structures appropriate to the internal representations may also be defined using standard data-defining tools. This method also provides the user with an appropriate set of selectors and constructors for use in writing the routines for operations on these structures, and the mode analyzer then checks to ensure that such operations are always applied to the correct data-structures. However, as this point is considered in some detail in the cited references we shall not repeat the discussion here.

Once such data-defining tools are available, even the task of defining an algebraic expression becomes much easier. Such a mode could be defined as follows:

```
MODE ALGEBRAIC = UNION(ATOM, ALGSTRUCT),
     ALGSTRUCT = STRUCT(LOP:LITERAL,ARGLIST),
       ARGLIST = LIST OF ALGEBRAIC;
```

Here *atom* and *literal* are the predefined system modes mentioned earlier, UNION constructs a new mode which because its first argument, ATOM, is self-identifying does not need an additional run-time tag, and STRUCT builds a structure with two fields, the first of which is selected by LOP and the second by ARGS. The mode of the second field, ARGLIST, is further defined as a list of algebraics, so that the standard list selectors HEAD and TAIL can access parts of that list. For example, given a (non atomic) algebraic expression defined in this manner, LOP will select its leading operator and HEAD ARGS the first argument of that operator. One might wish to go one step further and limit the literals allowed to be LOPs to a class so tagged, but this presents no difficulties.

At first sight, it looked as though the second task on our list would involve extensive programming. However, in looking at the structure of the existing REDUCE simplifier an interesting fact emerged. Like most LISP-based simplification programs, the top level function in this program has the name SIMP which then calls other functions with names like SIMPPLUS, SIMPTIMES, etc. on expressions with particular top-level operators. However, as these routines transform their arguments from one form to another, most of the analysis performed is semantic in nature. Since in the mode analyzing model such analysis is also performed by MEVAL, it became obvious that the main part of the work done by SIMP could be replaced by MEVAL and a few small auxiliary functions. In fact, in the new model an acceptable definition for SIMP of an expression is simply EVAL of MEVAL of the expression! Since a call to SIMP is often needed dynamically, the need to have MEVAL also available for dynamic use is therefore apparent.

This definition of SIMP has the advantage that the current body of code associated with SIMP and its auxiliary functions can be removed in favor of the more general mode analyzer. The task of adding an interface to a new internal form is therefore reduced to writing a straightforward function for converting a kernel to this form plus a few auxiliary functions concerned with the handling of

such things as unusual exponents. Some straightforward changes to the mode tables of the relevant operators can now make this new internal form the system default.

Another immediate advantage of this technique is that because the mode analyzer can be (preferentially) used statically as well as dynamically, we have the possibility of optimizing calculations further by doing as much of the semantic analysis as possible statically.

Consider for example the following program to compute the first eight terms in the famous f and g series:

```
DEPS := -SIG*(MU+2*EPS);
DMU  := -3*MU*SIG;
DSIG := EPS-2*SIG**2;

F1 := 1; G1 := 0;

FOR I := 1:8 DO BEGIN
   F2 := -MU*G1 + DEPS*DF(F1,EPS) + DMU*DF(F1,MU)
         + DSIG*DF(F1,SIG);
   G2 := F1 + DEPS*DF(G1,EPS) + DMU*DF(G1,MU)
         +DSIG*DF(G1,SIG);
   F1 := F2; G1 := G2 END;
```

Within the loop it is fairly easy for the mode evaluator to determine that EPS, MU, SIG, DEPS, DMU and DSIG are invariants. Furthermore, it knows from analysis prior to the loop that EPS, MU and SIG have (idempotent) mode *kernel*, and that the other literals have a value whose mode is *internal*, in this case an internal form for polynomials (since no division occurs). Finally since F1, F2, G1 and G2 are assigned within the loop to expressions with top level internal operators, they too are of mode *internal*. So a term like MU*G1 can be replaced by the form MULTKF(MU,G1), where MULTKF is a function for multiplying a kernel by an internal form. If such a function does not exist, coercion would transform the kernel to an internal form, giving say MULTF(K2F(MU),G1). Finally, the analyzer can recognize that K2F(MU) has a constant value and therefore put that value in directly. One can similarly analyze each expression in the loop and so produce an optimized form for the loop before it is evaluated.

Another way to optimize such a calculation would be to declare EPS, MU and SIG as *kernels* and the remaining literals as *polynomials* by saying

```
DECLARE EPS,SIG,MU:KERNEL,
        DEPS,DSIG,DMU,F1,F2,G1,G2:POLYNOMIAL;
```

Given that *polynomial* was the mode of the internal form for polynomials such a declaration would cause all expressions involving these literals to be treated in such a mode, and incidentally give a more expert user the direct access to that internal mode we mentioned earlier as desirable in some cases. This would clearly be the most efficient way in general of treating such problems, but it is interesting to note in this case, that even with no declarations at all, the system was able to do statically all the necessary optimization itself.

Compiler experts will recognize that such optimizations are little different in nature from those one might apply to any program, and this is probably no accident since our ideas have clearly been influenced by some recent work on designing a portable compiler for the REDUCE embedding language LISP (18). Eventually we hope to use the general analysis provided by the mode analyzer as a means of improving the compilation process still further, but this remains for the future.

The last task to be faced in introducing a new internal mode is perhaps the messiest, namely the conversion of an internal form to an algebraic expression, or printing it directly in such a form. The mode evaluator cannot help here because the syntax of the internal representation is unlikely to be of a form understood by MEVAL. So these routines must be written directly. However, this turns out to require only a page or two of code in most cases and the task is again eased by using the appropriately named selectors and constructors introduced in defining the different data-sturctures.

The most important thing, of course, is that our analysis has localized in a fairly modular way the necessary additional code to be written when introducing a new mode. One can therefore introduce a new representation confident that no surprises will occur when the internal form is used.

5. Conclusion

We have described in this paper a new model for simplification in REDUCE which allows for far more flexibility and ease of description than previously. At the same time, the use of general mode analyzing concepts has not only helped with this description but also allowed for an implementation with such obviously desirable properties as greater robustness and efficiency, and more ease of maintenance by the modular localization of internal representations. The result should be a system with far greater user appeal than the current version, which, in addition to providing users with a well defined natural interface, will also allow them easy access to the internal representations if they so desire.

6. Acknowledgment

The author is grateful to the Centre de Physique Théorique, CNRS, Marseilles, France, for its hospitality during the period when this paper was written.

7. References

(1) BROWN, W.S., *On Computing with Factored Rational Expressions, SIGSAM Bulletin,* ACM, New York, *8* (1974) 27-34.

(2) HALL, A.D., *Factored Rational Expressions in ALTRAN, SIGSAM Bulletin,* ACM, New York, *8* (1974) 35-45.

(3) HOROWITZ, E. and MUSSER, D.R., *The Synthesis and Use of Algebraic Specifications of Data-Structures,* USC Preprint 1975.

(4) HEARN, A.C., *A Mode Analyzing Algebraic Simplification Program, Proc. ACM 74,* ACM, New York (1974) 722-724.

(5) WEGBREIT, B., *Comm. ACM 17* (1974) 251-264.

(6) LINDSEY, C.H., and Van der MEULEN, S.G., *Informal Introduction to ALGOL 68,* North-Holland (1973).

(7) WIRTH, N., *Acta Informatica 1* (1971) 35-63.

HOARE, C.A.R., *Notes on Data Structuring* in *Structured Programming,* Academic Press (1972).

(8) NORDSTROM, M., *A Parsing Technique, Univ. of Utah Comp. Phys. Operating Note No. 12* (1973).

(9) GRISS, M.L., *The Definition and Use of Data-Structures in REDUCE, Proceedings of SYMSAC 76,* ACM, New York (1976).

(10) MOSES, J., *Comm. ACM 14* (1971) 527-537.

(11) TOBEY, R.G., BOBROW, R.J. and ZILLES, S.N., *Automatic Simplification in FORMAC, Proc. AFIPS 1965 Fall Joint Comput. Conf., Pt. 1,* (1965) 37-52.

(12) MOSES, J., *Comm. ACM 14* (1971) 548-560.

(13) JENKS, R.D., *The SCRATCHPAD Language, SIGPLAN Notices,* ACM, New York, *9* (1974) 101-111.

(14) HEARN, A.C., *The Problem of Substitution, Proc. IBM Summer Institute on Symbolic Math. by Computer, IBM Prog. Lab. Rep. No. FSC-69-0312* (1969) 3-19.

(15) HEARN, A.C. *The REDUCE Program for Computer Algebra, Proc. 3rd Colloq. on Advanced Comp. Methods in Theor. Phys.,* C.N.R.S., Marseilles, (1973).

(16) SHTOKHAMER, R., *Simple Ideal Theory, Some Applications to Algebraic Simplification, Univ. of Utah Comp. Phys. Report No. UCP-36* (1975).

(17) STOUTEMYER, D., *Automatic Error Analysis Using the Computer Symbolic Language REDUCE, Trans. Math. Software* (to be published).

(18) GRISS, M.L., and HEARN, A.C., *A Portable LISP Compiler, Univ. of Utah Comp. Phys. Report No. UCP-47* (1976) (to be published).

The Definition and Use of Data Structures in Reduce *

Martin L. Griss
University of Utah

Abstract

This paper gives a brief description and motivation of the mode analyzing and data-structuring extensions to the algebraic language REDUCE. These include generic functions, user defined recursive data structures, mode transfer functions and user modifiable automatic coercion. A number of examples are given to illustrate the style and features of the language, and how it will aid in the construction of more efficient and reliable programs.

1. Introduction

REDUCE [1] is a special-purpose program for computer algebra and symbol manipulation. At the same time, it is an extensible, general-purpose language for experimenting with algebraic algorithms and language features to aid the system user.

REDUCE currently operates in one of two modes, ALGEBRAIC or SYMBOLIC, corresponding to the data-types "algebraic expression" or "LISP S-expression." The typical user writes commands and procedures in ALGEBRAIC mode, invoking algebraic simplification routines. More sophisticated users can also use SYMBOLIC mode; in fact, the whole of REDUCE is written in this symbolic subset, an ALGOL-like language that translates directly to LISP. This self description enhances extensibility and portability.

The program consists of many procedures, operating on some internal data-structures. Operators are expressed in terms of the ubiquitous CAR, CDR, and CONS of the underlying LISP S-expressions. For example, there are polynomials with integer or rational coefficients, matrices with integer, rational or polynomial elements, rational functions and others. Each of these has an associated set of functions to access, manipulate, simplify or print all or parts of the corresponding structure. Each of these functions has its own unique name, although they accomplish rather similar things for the different data-types. In fact, even the structure of many of these functions is the same, but this is obscured by a plethora of names, and minor differences. Some of these functions have been merged, with alternate segments selected by run-time tests. This has made it difficult to comprehend or modify the underlying algorithm.

After examining the REDUCE source the sophisticated user can design and manipulate his own data-structures--for example, sparse matrices with elements that are polynomials with complex coefficients. To interface his package naturally requires considerable familiarity with the whole system. The casual user confuses the numerous special function names, and the specific combinations of CAR's and CDR's required to select portions of his own or system data-structures to pass between the various routines. In fact, experience indicates that most of our programming errors result from the misapplication of combinations of CAR's and CDR's, or from type-mismatches.

Loos [2] has stressed the importance and utility of writing algebraic algorithms in REDUCE using abstract selectors and constructors for the appropriate data-structures [3]. The current version of REDUCE has a simple macro facility which can be used to map meaningful algebraic selectors such as LT, LC, MVAR or RED to combinations of CAR and CDR, while constructors such as ADD or MULT become CONS or LIST. Using this facility, algorithms become more readable and easier to debug. However, the current system gives the user no assistance in ensuring that a selector is applied to the correct data structure. Finally, a given "abstract" selector can map to only a single specific function, and the specific names of many different functions must still be remembered by the user.

A version of REDUCE under development [4] extends these mnemonic features, permitting a much tighter control over the functions that can be applied to various data-types, ensuring correct and efficient usage. This is done by associating a "mode" with each distinct data-type, and using these to select automatically the correct function to apply to a given set of arguments, making it easier to add new packages which use the full power of the existing system. In addition, user programs have less change of error because they will have a more straightforward relationship to the natural mathematical language used in defining the problem.

*Work supported in part by the National Science Foundation under Grant No. GJ-32181.

The ideas of mode-analysis and user defined data-structures are not new. The present implementation [4] has been influenced by features found in PASCAL, ALGOL-68 and EL1 [5]. However, our aim of providing easy, safe and modular access to portions of the whole system without requiring the experience or sophistication of a system implementor, leads to a different emphasis. The casual user works in an interactive environment, in which defaults are required to avoid extensive declarations.

We are incrementally extending the current REDUCE system with mode-analysis and data-structuring "packages" written in REDUCE, rather than using or designing a completely new language for three reasons. They are:

i) REDUCE is a fairly portable system, existing within different environments on a number of different machines, and has a sizable user community.

ii) Also, we want to retain as much of the existing system as possible, so that most previously written programs will still work.

iii) REDUCE already provides all the power of LISP, plus a large body of tested procedures implementing algorithms for computer algebra and symbolic manipulation. (It would be a non-trivial feat indeed to rewrite all of REDUCE in PASCAL or ALGOL-68, for example.)

Section 2 describes the overall picture of mode analysis, generic functions, and mode transfer with coercion. We use a PASCAL-like [5] syntax whenever the original REDUCE ALGOL-60 notation is inappropriate. Section 3 gives more detail on the user data definition facilities. Section 4 describes some larger examples, including a form of "uniform referent" to aid in refinement of an abstraction. Finally we consider possible changes and further work in Section 5.

2. Mode Analysis

i) Modes.

A number of "primitive modes are predefined in REDUCE, and additional modes can be added. These modes flag data and procedures so that a mode-analysis phase following parsing can check the compatibility of functions with their arguments. Modes which are currently predefined include: the conventional BOOLEAN, INTEGER, REAL, SYMBOLIC, and ARRAY and a set of algebraic modes: KERNEL, POWER, TERM, POLYNOMIAL, RATIONAL, MATRIX and ALGEBRAIC. The latter have been declared to act as an interface to the many data-structures and procedures already present in REDUCE.

The mode of any expression is determined by a simple bottom up analysis, starting with the explicit or declared modes of constants and variables. The argument modes presented to a given function or operator are checked against a table of permissable argument sets: an exact match selects the specific function to use and the result mode to pass upwards; the absence of a match signals an error or could invoke coercion.

ii) Generic Functions

Because a number of alternative argument mode sets can be associated with a given function or operator name, a function such as PLUS becomes "generic" or polymorphic. The mode analysis phase selects the specific instance required for the given arguments. Thus PLUS will map to IPLUS for *integer* + *integer*, RPLUS for *real* + *real* or ADDF for *polynomial* + *polynomial*. Previously this selection would be done at run-time or by the user inserting the correct function by hand. A new meaning for an operator is easily added, by simply defining a procedure with the appropriate modes as arguments.

A particular generic name can become "overworked," obscuring the abstraction it is supposed to represent in the source. REDUCE, therefore, permits the definition of new infix operators from combinations of special symbols. For example, ".+" may be defined to suggest some similarly to the operator "+".

iii) Mode Hierarchies and Transfer Functions

In the above, each mode is considered entirely independent, without any relationship to any other. However, in many situations, particularly for algebraic manipulation, sets of modes have quite natural, hierarchical relationships: *integers* may be considered as special cases of *real* or *rational numbers*; *polynomials* are special instances of *rational functions*; and so. These "lower" modes are distinguished because of additional features, while still sharing many properties with the more general "higher" mode. These modes can often be cheaply converted into their "higher" mode using a unary transfer function, saving the additional definition of operations on "mixed" argument sets. An extremely useful convention for naming a transfer function from mode "M1" to "M2" is to use the target mode name "M2" generically: REAL(I), would convert an *integer* I to a *real*.

Thus *integer* + *real* can be done as RPLUS(REAL(*integer*), *real*), if desired. This convention also agrees rather nicely with the convention that the mode name may also be used as a constructor. For example, defining a complex mode:

```
MODE COMPLEX = STRUCT(RL:REAL, IM:REAL);
```

we construct the *complex* "1+2i" using COMPLEX(1,2). Here, COMPLEX is a generic function, and if we wish to consider a *real* as a *complex* with a 0 imaginary part, we define the transfer function COMPLEX(*real*): (with one argument):

```
COMPLEX PROCEDURE COMPLEX(R);
   DECLARE R:REAL;
      COMPLEX(R, 0.0);
```

Finally, if we want an *integer* to become a *complex* via *real*, we can simply write:

```
COMPLEX PROCEDURE COMPLEX(I);
   DECLARE I:INTEGER;
      COMPLEX(REAL I, 0.0);
```

which defines yet another meaning for the generic function COMPLEX(...).

iv) Coercion.

In a situation with a large number of inter-related modes, it is tedious to continually insert the requisite transfer functions, even though the convention is quite mnemonic. This requirement is particularly annoying in an interactive algebraic system. We therefore have provided a simple coercion mechanism [4] to automatically insert transfer functions. An allowed coercion between a pair of modes is entered into a graph, with the name of an associated transfer function. When an exact match for one of the argument sets of a function fails, an attempt is made to find at most a single coercion of each argument to match an allowed argument set.

Frequently coercions exist to permit a number of different modes to be chosen to perform a given operation. For example, *integer* + *real* could be done as *real* addition, or as *complex* addition, or even more expensively as *polynomial* addition. To overcome this ambiguity, a weight is associated with each link in the graph and the argument set with the least total weight of required coercion is selected.

This coercion mechanism is entirely under user control: links may be added, or removed, the weights or functions redefined at any time, and the entire graph or portions of it displayed. The system initially has some coercions predefined, in order to make algebraic manipulation most natural for the casual user, and to permit him to define his own modes that attach to the rest of the system with a minimum of effort.

3. Data-Structuring Facilities

More complicated data structures are built using mode generators. Each generator takes other modes or generators as arguments, producing specific functions for the correct modes and establishing the entries to the appropriate generic functions.

To introduce two new modes E1 and M2 we write:

```
MODE E1,M2;
```

DECLARE is used to declare the modes of local or global variables or procedure arguments, and to ensure that variables are correctly initialized with a value of the corresponding mode:

```
DECLARE R,S,V:EL,BB,XX:M2,I,J:INTEGER;
```

The format of a procedure definition is:

```
vmode etype PROCEDURE fn(args);

    [DECLARE args;] <body>;
```

where "etype" is the REDUCE evaluation method (EXPR, FEXPR, SMACRO or NMACRO) and "vmode" is the result mode of the procedure. The defaults "etype" = EXPR and "vmode" = ALGEBRAIC are used if either "etype" or "vmode" are omitted.

The argument modes default to "vmode," unless overriden by [DECLARE args;].

MODE also accepts more general constructions:

Mode	Description
(atom,...,atom)	Enumeration of values
LISTOF	elements of same mode, use CAR, CDR, etc.
ARRAY	elements all of same mode, use standard indexing
STRUCT	individual elements have selectors
UNION	combine a number of alternative modes, possibly tagged

The full mode declaration syntax is given in the appendix, and the following brief descriptions explain interesting features.

a. rmode=(atom,...,atom). Defines a mode consisting of a finite enumeration of explicit constants. A BOOLEAN predicate *rmodeP* is generated, to test if a given SYMBOLIC or INTEGER constant belongs to *rmode*. Transfer functions named *rmode*, from SYMBOLIC or INTEGER are also generated. These either return a value of *rmode*, or invoke a run-time error signal. E.g.

```
MODE FUNNY = (FALSE, MAYBE, TRUE);
```

generates a predicate FUNNYP(*funny*) and transfer functions FUNNY(*symbolic*) and FUNNY(*integer*).

b. ellst-LISTOF el. Defines a LISP-like list, with the standard "generic" list selectors and other functions made available: LIST, NULL, EQ, CAR, CDR, COPY, CONS, etc.

A useful syntactic construct for this mode is:

```
FOR EACH el { IN } ellst DO stmt;
            { ON }
```

which will check that "ellst" is a LISTOF mode, and will assign to "el" the correct element mode within "stmt." It is used to sequence through successive elements (or CDR's) of a list. It may be thought of as a macro for a WHILE loop that uses the generic functions NULL(*ellst*), CAR(*ellst*) and CDR(*ellst*) in the obvious way.

c. snam=STRUCT(sel1;mspec1,...) [:consnam]. Constructs a structure with element selectors mapping to some combinations of CAR and CDR and constructor mapping to LIST (or CONS if only

two elements are present). If "consnam" is omitted, the mode name "snam" will be used as the generic name of the constructor. For example, LISP-like dotted pairs might be defined as:

```
MODE DPTR =
   STRUCT(HEAD:SYMBOLIC,TAIL:SYMBOLIC):CNS,
```

This defines selectors HEAD(*dtpr*) and TAIL(*dtpr*), mapping to CAR and CDR and a constructor CNS(*symbolic*, *symbolic*) mapping to CONS. Variables declared to be DTPR's are initialized to (NIL.NIL).

If a system flag LHS is ON at the time the mode is defined, these selectors may also be used to deposit into a selected field of a STRUCT. (A LOAD/STORE "duality".) This use is somewhat controversial, since it does in fact invoke RPLACA and RPLACD, LISP operations which are difficult to use without introducing errors. However, REDUCE or LISP programmers use RPLACA and RPLACD in the interests of efficiency: by using the mode-checking facilities, we can make these usages somewhat safer. We do this by ensuring that we cannot apply RPLACA or RPLACD indiscriminately, but only to selected data-types, which will have been correctly initialized at declaration time. If XY is a DTPR (X.Y), then

```
X:=HEAD(XY);
```

selects the first field while

```
HEAD(XY):='Z;
```

will actually invoke

```
RPLACA(XY, 'Z);
```

d. unam=UNION([tag1:]mspec1,[tag2:]mspec2,...)
Defines a discriminated union. A particular item of mode *unam* consists of a dotted pair (Tag.Body), where the tag can be examined at run-time to distinguish among the modes mspec1, mspec2, etc. If any tag is omitted, the associated mode name is used instead. We were careful to keep the tag "conceptual," in that the user cannot explicitly access the tag, nor change the "tag" and "body" parts independently. For each element (tag.body) a generic predicate "tagP"(*unam*), selector "tagB"(*unam*) and a transfer function "unam"(*bodymode*) are constructed. For example, the definition:

```
MODE NUMBER = UNION(FIX:INTEGER, FLOAT:REAL);
```

will create BOOLEAN predicates FIXP(*number*) and FLOATP(*number*) an INTEGER selector FIXB(*number*) and a REAL selector FLOATB(*number*), and transfer functions (constructors) to mode NUMBER: NUMBER(*integer*) and NUMBER(*real*).

Since the tag is never directly accessed, certain self-identifying modes can be efficiently represented without any tags at all. E.g.

```
MODE NUMBER = UNION(INTEGER,REAL);
```

will simply use the system supplied INTEGERP and REALP, with INTEGERB(*number*) and NUMBER(*integer*) being "do-nothing" functions.

We will define recursive data-types [6] by using the convention that variables of UNION mode are initialized to the first element of the UNION, and that it should not be involved in the recursion. E.g. the recursive mode SYMBOLIC might be defined as

```
MODE SYMBOLIC = UNION(ATM:ATOMIC,NONATM:DTPR);
```

with DPTR as declared above, and ATOMIC some built-in mode, initialized to NIL. Thus, DECLARE X,Y:SYMBOLIC; will initialize X and Y to NIL, as desired.

These modes are best used with a CASE construct that makes explicit the set of run-time tests it represented:

```
CASE m[:=stmt] OF
   T1:stmt1;
   T2:stmt2;
      .
      .
   Tn:.......
   ELSE:.....;
END;
```

Here the T_i represent the tags actually used. Within "$stmt_i$" the variable "m" will be bound to "$tag_iB(...)$" and so will behave as if it had "$mode_i$" further hiding the tag's existence even more. Thus in the segment

```
DECLARE N:NUMBER, I:INTEGER, R:REAL;
CASE N OF
   FIX: I:=N;
   FLOAT: R:=N;
END;
```

the tag associated with N is correctly "stripped-off" within each branch of the CASE.

Two dimensional CASE statements

```
CASE m1[:=stmt1], m2[:=stmt2] OF
   T1, T2:stmt12;
   .
   .
END;
```

are extremely useful in an algebraic context, as will be seen in a subsequent example.

4. Comments and Examples

These mode-generators may be combined to build structures that refer to modes not yet defined. (This enables the definition of recursive data-structures [6]. The definition, of course, has to be completed before its first use in a DECLARE statement.)

a. A "complex" mode with integer elements.

```
MODE COMPLEX = STRUCT(RL:INTEGER, IM:INTEGER);
```

An object of mode complex will be initialized to (0 . 0) when declared, and may be constructed using COMPLEX(i,j):

```
DECLARE C1, C2:COMPLEX;
   C1:=COMPLEX(2,5);
```

The parts are selected using RL, or IM and may be changed using these selectors on the left hand side:

```
RL(C2): = IM(C1) + 5;
```

which sets C2 to be (10 . 0). The "+" of course maps to IPLUS (*integer*, *integer*);. We now define a few "COMPLEX" operations:

```
COMPLEX PROCEDURE C1 + C2;
   COMPLEX(RL C1 + RL C2, IM C1 + IM C2);

COMPLEX PROCEDURE -C;
   COMPLEX(-RL C, -IM C);

INTEGER PROCEDURE MODSQ C;
   DECLARE C:COMPLEX;      % Override default
   RL(C)**2 + IM(C)**2;
```

If we now write "C1 + 3;" we get a mode-mismatch error, since neither IPLUS nor PLUS(*complex*, *complex*) may be used. First we define the transfer function:

```
COMPLEX PROCEDURE COMPLEX I;
   DECLARE I:INTEGER;
      COMPLEX (I,0);
```

Now "C1 + COMPLEX(3)"; is of course legal. To define coercion, we execute

```
COERCE INTEGER COMPLEX;
```

so that finally "C1 + 3"; behaves as desired. The full form of COERCE is "COERCE M1 M2 wt trans"; with "wt" defaulting to 2, and "trans" to M2.

b. Definition of a polynomial data-type. In many cases, the meaningful constructor is not the modename, and "consnam" is used instead. The declarations

```
MODE POLY=UNION(INT:INTEGER,POL),
   POL=STRUCT(LT:TRM,RED:POLY):.+
   TRM=STRUCT(TPOW:PWR,TC:POLY):.*,
   PWR=STRUCT(PVAR:KERNEL,PDEG:INTEGER):.**;
```

define the selectors and constructors for a fairly complicated recursive polynomial structure, somewhat simplified from that actually used in REDUCE. ".+" is a "static" constructor that "adds" a TRM to a POLY to produce a POL (POL's are known to have structure, and are not the trivial case INTEGER).

We define an additional pair of selectors LPOW and LC used to access the leading power and leading coefficient of a POL as SMACRO's:

```
PWR SMACRO LPOW P;
   DCL P:POL;
      TPOW LT P;

POLY SMACRO LC P;
   DCL P:POL;
      TC LT P;
```

(SMACRO defines a simple substitution macro and these procedures expand in line to (CAR(CAR P)) and (CDR(CAR P)) for efficient execution. DCL is an abbreviation for DECLARE.)

We can now define POLY addition and negation quite simply by:

```
POLY PROCEDURE B + P;
   DCL B:INTEGER;
   CASE P OF
      INT:POLY (B +P);
      POL:POLY (LT P .+ (B + RED P));
   END;

POLY PROCEDURE -P;
   CASE P OF
      INT:POLY(-P);
      POL:POLY((LPOW P .* -LC P) .+ (-RED P));
   END;
```

The "POLY" within the alternates of the case statement is used to produce a POLY value from either an INTEGER or a POL; if coercions

```
"INTEGER" → "POLY",
```

and

```
"POL" → "POLY"
```

are defined by

```
COERCE INTEGER POLY;
```

and

```
COERCE POL POLY;
```

this transfer function would be inserted automatically.

The first routine is rather simple, since it adds an INTEGER to a POLY; to add two POLY's we first define a canonical ordering operation

```
BOOLEAN PROCEDURE P1 > P2;
   DCL P1,P2:PWR;
      IF PVAR P1=PVAR P2 THEN PDEG P1 > PDEG P2
         ELSE PVAR P1 > PVAR P2;
```

between two PWR's, in terms of a system defined function

```
BOOLEAN PROCEDURE V1 > V2;
   DCL V1, V2:KERNEL;
      ------;
         etc.
```

for ordering KERNEL's (generalized variables).

Finally, using *integer* + *poly* and the ordering function, *poly* + *poly* becomes a simple merge with the addition of compatible terms:

```
POLY PROCEDURE P + Q;
   CASE P,Q OF
      INT,INT:POLY(P + Q);
      INT,POL:POLY(LT Q .+ (P + RED Q));
      POL,INT:POLY(LT P .+ (Q + RED Q));
      POL,POL:POLY(IF LPOW P > LPOW Q
                     THEN LT P .+ (Q + RED P)
         ELSE IF LPOW Q > LPOW P
                THEN LT Q .+ (P + RED Q)
         ELSE (LAMBDA (CC,RR);
            IF CC=0 THEN RR
               ELSE (LPOW P .* CC) .+ RR
                  (LC P + LC Q, RED P + RED Q));
   END;
```

c. "Uniform referent" features in REDUCE. Selectors, function calls and array access all use the same functional notation, "f(arg$_1$,arg$_2$,..arg$_N$)," so that constructs may be redefined with a minimum of subsequent source changes; PROCEDURES may be replaced by SMACRO's for in-line expansion and arrays may be replaced by STRUCT's or PROCEDURES.

Consider an integer array, H:

```
DECLARE H:ARRAY(1000) OF INTEGERS;
```

The jth element is accessed, and assigned by

```
J:=H(J);     H(J):=integer expression
```

wherever necessary.

If at some later stage we decide that there is insufficient array space for 1000 locations or discover that H is rather sparse, we can refine our initial abstraction using simple chained hashing:

```
MODE HPR=STRUCT(IND:INTEGER, VAL:INTEGER);
   HLST=LISTOF HPR;

DECLARE HH:ARRAY(10) OF HLST;
```

We now define new access and assignment functions H:

```
INTEGER PROCEDURE H(I);
   BEGIN DCL HL:HLST;
      HL:=HH(REMAINDER(I,10));  %Select bucket;
   A:IF NULL HL THEN RETURN 0;  %Default value;
      IF IND CAR HL=I THEN RETURN VAL CAR HL;
      HL:=CDR HL;
      GOTO A;
   END;

INTEGER PROCEDURE H(I):=J;
   BEGIN DCL HL1, HL2:HLST;
      HL1:=HL2:=HH(REMAINDER(I,10));
   A:IF NULL HL1 THEN GOTO B; %None found;
      IF IND CAR HL1=I
        THEN RETURN(VAL CAR HL1:=J);
      HL1:=CDR HL1;
      GOTO A;
   B:HL1:=HPR(I,J) . HL2;
      HH(REMAINDER(I,10)):=HL1;
      RETURN J;
   END;
```

Notice the importance of being able to define "H(I):=J"; as a generic instance of SETQ. Further refinements are possible, such as keeping the buckets ordered, checking the range of I, removing elements for H(I):=0; and so on. Nevertheless, no other source changes will be required.

d. Error detection and production of more efficient code. Many errors that used to be detected only at run-time (in a mode-less LISP environment) are now seen as mode-mismatch or mis-application-of-selector errors by the mode-analyzer. This permits diagnostic information to be more complete, referring to the meaningful abstract or generic names, before the renaming to internal functions.

By removing the many run-time mode tests of the old system, significant gains during interpretation (especially of loops) arise. The more efficient LISP produced by the system can then be further compiled into efficient machine code, if desired, using a portable compiler written in REDUCE itself [7].

This facility can also defer to the systems implementor (or the user in the next level of refinement) a choice of data-structure to implement a STRUCT or UNION, without formally changing previously written procedures.

For example, the simulation of a PDP-10 machine word:

```
MODE BIT=(0,1),WORD=ARRAY(35) OF BIT;
```

is being considered in the context of rewriting an entire LISP interpreter in REDUCE itself, so as to easily experiment with compacting garbage collectors and efficient language constructs most natural to algebraic manipulation. At a later stage, the functions generated to access individual *bits* of a *word* may have to be hand-coded in machine language but procedures using them need not be changed.

5. Conclusions

This facility helps the programmer to think about his data-structures in terms other than the LISP S-expressions out of which they are composed. The tighter control over access to the data-structures via meaningful selectors and constructors whose consistent and correct usage is checked by the system, makes the implementation of new algorithms cleaner and less error prone. The mapping of single "generic" operators to specific data-dependent functions sheds new light on the clarity and efficiency of previous implementations.

A large number of procedures are now being rewritten to test the utility of these ideas. In particular, the automatic coercion is likely to be the most disturbing to "strong typing" advocates. We will have to determine whether the possible ambiguities outweigh the advantages of "natural" style.

6. Acknowledgments

It is a pleasure to thank Dr. A. C. Hearn and Dr. M. Rothstein for many useful discussions.

7. References

[1] HEARN, A.C., "REDUCE 2 - A System and Language for Algebraic Manipulation," Proc. of Second Symposium on Symbolic and Algebraic Manipulation, International Hotel, Los Angeles (1971) 128-133.

HEARN, A.C., "REDUCE 2 User's Manual," 2nd Edition, University of Utah Computational Physics Group Report No. UCP-19, March 1973.

[2] LOOS, R.K., "Algebraic Algorithm Descriptions as Programs," SIGSAM Bulletin ACM 23 (1972) 16-24.

[3] WEGNER, P., "Programming Languages, Information Structures and Machine Organization," McGraw-Hill (1971) 217.

LANDIN, P.J., "The Mechanical Evaluation of Expressions," Computer Journal (1964).

[4] HEARN, A.C., "A Mode Analyzing Algebraic Manipulation Program," Proc. of ACM 74, San Diego (1974) 722-724.

[5] WIRTH, N., "The Programming Language Pascal," Acta Informatica 1, (1971) 35.

HOARE, C.A.R., "Notes on Data-Structuring," in "Structured Programming," Academic Press (1972).

LINDSEY, C.H. and S.G. van der MEULEN, "Informal Introduction to ALGOL-68," North-Holland (1973).

WEGBREIT, B., "The Treatment of Data-Types in EL1," CACM v5 (1974) 251-264.

[6] HOARE, C.A.R., "Recursive Data-Structures," Stanford University Preprint (1973).

[7] GRISS, M.L. and HEARN, A.C., "A Portable LISP Compiler," University of Utah Computational Physics Group Report No. UCP-47 (1976).

8. Appendix

SYNTAX OF MODE DEFINITION

```
mode-defn::=MODE mspec [,mspec,mspec.....];

mspec::=modnam | modef | modnam = mspec

modnam;::=identifier

modef::=rangedef | listdef | arraydef |
     structdef | uniondef

rangedef::=(atom[,atom...])

listdef::=LISTOF mspec

arraydef::=ARRAY(limit[,limit...]) OF mspec

limit::=integer

structdef::=STRUCT(sel:mspec [,sel:mspec...])
     [:consnam]

consnam::=identifier

sel::=identifier

uniondef::=UNION([tag:]mspec,[tag:]mspec [.....])

tag::=identifier
```

A Pattern Compiler

Richard D. Jenks
Mathematical Sciences Department
IBM T. J. Watson Research Center
Yorktown Heights, New York

ABSTRACT. A pattern compiler for the SCRATCHPAD system provides an efficient implementation of sets of user-defined pattern-replacement rules for symbolic mathematical computation such as tables of integrals or summation identities. Rules are compiled together, with common search paths merged and factored out and with the resulting code optimized for efficient recognition over all patterns. Matching principally involves structural comparision of expression trees and evaluation of predicates. Pattern recognizers are "fully compiled" if values of match variables can be determined by solving equations at compile time. Recognition times for several pattern matchers are compared.

1. **INTRODUCTION.** One of the most useful features of a system for symbolic mathematical computation is a pattern matching facility which allows user-defined pattern-replacement rules to be incorporated automatically, and without programming, into the general system simplifier for a specific application. Unfortunately this general approach of adding user-defined transformations is known to frequently cause serious degradation in performance when a large number of patterns is present since all subexpressions must be checked for a possible match.

The "ruleset" concept of SCRATCHPAD [3] allows a user to capture the set of pattern-replacement rules which exist at any instant in time, to give this set a name, and then later selectively invoke this set of rules simply by reference to this name. Rules are first created through commands all of which may be regarded as pattern-replacement rules [3], e.g.

(1) $\int_x x^m \times \cos(x) = \cdots \quad (m \in (2,3,...))$

Once all constituent rules of the ruleset have been introduced, a ruleset named *trigint* is created by issuing the command

trigint= ' rules

The rules are then cleared and the session continues. At some later time, the user may issue a command

u,trigint (resp. u,+trigint)

which calls for the evaluation of u with the current environment replaced by (resp. augmented by) the ruleset *trigint*. Rulesets can be augmented or diminished by other rulesets; they also can be saved on secondary storage where they may be retrieved in a later session.

This paper describes a pattern compiler for rulesets, that is, a function which will transform a set of patterns into a compiled LISP function to "recognize" whether an argument expression represented in LISP prefix form is matched by a pattern of the set. The resultant compiled code is considerably efficient as a result of a design based on the following ideas:

(a) The order of terms in products can often be fixed by a suitable definition of ordering on expressions. For example, given a suitable ordering, all instances of (1) will have the 'x^m' term appearing before the 'cos(x)' term. As a result, all expressions of the class will then have the following LISP prefix form as internal representation:

(2) (∫ (× (↑ x m)(COS x) x)

When the order of terms of products cannot be fixed, terms can nevertheless be arranged into groups, where each group has a fixed ordering with respect to the others and where the order of terms within a group can be determined by a run time search resulting in a possible permutation of terms.

(b) General patterns can usually be split into more specific patterns where matching can be performed by structural examination of candidate expressions and by evaluation of predicates, in particular without dynamic searching or algebraic computation. For example, all instances of (1) with m instead satisfying $m \in (0,1,\cdots)$ has three subcases: m=0, m=1, and $m \in (2,3,\cdots)$ each with structurally different internal representations. Having determined that an expression has representation (2), one need only test that an exponent matching m is a positive integer.

(c) Recognizers for individual patterns can be uniformly generated then combined into a single decision tree in order to merge and factor out common code segments; this resulting tree can then be optimized for efficient recognition over all patterns of the set. For example, the members of the class of expressions

(3) $\int_x x^m \times \sin(x) \qquad (m \in (2,3,\cdots))$

all have an internal representation similar to (2) so that the recognizers for (1) and (3) can be essentially combined.

(d) Advantage can be taken of formal variables such as the variable of integration which appears in every term of the integrand but which cannot appear in any expression matched by other variables (e.g. m above).

A pattern will be said to "match" an expression if the expression can be obtained from the pattern by substituting "values" for variables into the pattern and simplifying. The classes of patterns which are fully compiled by this method are those for which these substitution values can be determined by solving linear equations at compile time. When this is not possible and when other techniques fail, a pattern is partially compiled with calls inserted to a general pattern matcher as needed.

Sections 2-6 systematically lead to the description of an algorithm in Section 7 for the translation of a given pattern into a LISP function which will recognize all instances of the pattern. The algorithm is naturally described by a walk down a "recognition tree" (Section 6) which is easily derived from a "pattern tree" (Section 5). In section 8, we sketch the entire process of compiling ruleset patterns. In section 9, we give a preliminary comparison of timings with this and two other pattern matchers. Section 10 compares our approach with that of the MACSYMA pattern matcher and presents our conclusions.

2. **EXPRESSIONS.** By an *expression* e we mean any identifier, integer, or *operator form*: (OP $a_1 \cdots a_n$) where OP is an operator (called the *main operator of* e) and $a_1,\cdots,a_n$ are expressions (called the *argument expressions of* e). By an *indexed operator form* we mean an operator form where OP is ∫,Σ, etc. for which one of its arguments is a formal identifier called an *index* (the variable of integration, summation index, etc.). All identifiers other than indices are regarded as *variables* which range over sets of values described by predicates over expressions.

We assume expressions are *normalized*, that is: (i) algebraic expressions are written solely in terms of +,×, and ↑ operations (i.e. quotients are replaced by exponentials, differences and negatives are replaced by sums and products with −1); (ii) subexpressions are algebraically simplified and factored with respect to content, where the usual notion of the content of a polynomial is extended to rational functions and operator forms in an obvious way, e.g. the expression

(4) $$\int_x (1-2x)^n LI(2,2-1/x)$$

when normalized becomes

(5) $$(-1)^n \times \int_x (2x-1)^n LI(2,x^{-1}\times(2x-1))$$

(iii) expressions with symmetric main operators such as + and × have argument expressions ordered by $<\cdot$, where $<\cdot$ is any total ordering on expressions such that two distinct forms (OP $a_1 \cdots a_n$) and (OP $b_1 \cdots b_n$) having the same main operator OP and equal length n are ordered as follows: First, if OP is not symmetric, let (p(1),⋯,p(n)) denote the "branch order" for OP (see below). If OP is symmetric, we assume already $a_1 <\cdot a_2 <\cdot \cdots <\cdot a_n$, $b_1 <\cdot b_2 <\cdot \cdots b_n$, and let (1,2,⋯,n) denote its branch order. Then (OP $a_1 \cdots a_n$) $<\cdot$ (OP $b_1 \cdots b_n$) if $a_{p(1)} <\cdot b_{p(2)}$, or if $a_{p(1)}=b_{p(2)}$ then $a_{p(2)} <\cdot b_{p(2)}$, etc.

More generally, normalization may be specified by a ruleset-dependent set Φ of transformations which convert expressions into a suitable form for matching (for example, Φ may call for integrands to be factored). We always assume however that normalization will include the transformations (i)-(iii) above.

Remarks. All expressions denoting patterns as well as expressions to be subjected to matching will be assumed normalized. The removal of content from indexed operator forms eliminates extraneous detail from integrals, sums, etc. so that patterns can be applied directly.

The notion of "environment" is central to symbolic computation in general and to pattern matching in particular. An *environment* is a list of n ordered pairs, called *bindings*, $s_i \rightarrow a_i$, $1 \le i \le n$, which associate a "target" expression a_i with a "source" expression s_i. For convenience we abbreviate environment $(s_1 \rightarrow a_1, \cdots, s_n \rightarrow a_n)$ by $(s \rightarrow a)$. The environment at any fixed instant in time is called the *current environment*. By *evaluation*, we mean any expression-to-expression transformation μ which involves a systematic replacement of source expressions in the original expression by corresponding target expressions.

3. **PATTERNS**. By a *pattern* p, we mean a triple (s,t,c) consisting of (1) a set s of n *match variables* $s_1,\cdots,s_n$, $n \ge 0$; (2) a *template* t=t(s) as defined below; and, (3) a predicate on expressions called the *match condition* c=c(s). A template t is either an expression or else has the form ('OP' $t_1 \cdots t_n$) for some operator symbol OP (called the *main operator of* t) and templates (called the *argument templates of* t) $t_1,\cdots,t_n$, $n \ge 1$. A pattern p=(s,t,c) is said to *match* an expression μe if there exists an environment $(s \rightarrow a)_M$ such that c(a)=**true** and μ(e−t(a))=0. Such an environment $(x \rightarrow a)_M$ is called a *match environment* for p. The *match domain* of p is defined to be the set of expressions matched by p. By a *recognizer for pattern* p we mean any total function on expressions which maps an expression e into a match environment or NIL according as p does or does not match p. Any expression to which a recognizer for a pattern is applied is called a *subject*.

The special pattern ('POLY' k ⋯) is used to represent a polynomial in an index k, where ⋯ denotes a list of the coefficients of the polynomial up to the leading degree (always assumed non-zero). As an example, the pattern derived from (1) has match variable set s=(x,m), match condition: (m is integer ∧ m≤−2) (note: x is guaranteed to be an identifier), and template

(6) $$\int_x \text{poly}(x,2,-1)\uparrow m \times LI(2,x\uparrow(-1)\times \text{poly}(x,2,-1)$$

with all operators quoted. We also assume patterns are *normalized*, that is: (i) the expression formed from the template by removing quotes from operators and replacing terminal expressions by unique identifiers is normalized and has content equal to one; (ii) terminal expressions in the template are normalized; (iii) match condition c is expressed in a conjunctive normal form, e.g. in the form ((r ∨ ⋯) ∧ ⋯)) where each r has one of the three forms: e<0, e=0, or e>0 for normalized e; (iv) Templates with a symmetric main operator have arguments templates ordered by $<\cdot$ which is defined as follows for two patterns $p_1=(s,t_1,c)$ and $p_2=(s,t_2,c)$ with distinct templates t_1 and t_2 but with identical match variable set s and match condition c. Pattern $p_1 <\cdot p_2$ if for each match environment $(s \rightarrow a)_M$: $t_1(a) <\cdot t_2(a)$. If, however, there exist match environments $(s \rightarrow a)_M$ and $(s \rightarrow b)_M$ such that $t_1(a) <\cdot t_2(a)$ yet $t_2(b) <\cdot t_2(b)$ then we say $p_1 =\cdot p_2$. The main operator of ('POLY' ⋯) is either +, ×, or ↑, thus is given special treatment.

Remarks. The key reason for introducing quoted operators is to distinquish between "structural" and "algebraic" matching. Template x'↑'y matches only those subjects with an explicit exponentiation operator, e.g. 2↑n with $(x \rightarrow 2, y \rightarrow n)_M$, but not $8=2^3$ (which is not normalized) or $4^u=2^{2u}$ (which requires a special Φ). If x and y appear more than once in a template and can be determined by solving equations, then an unquoted operator ↑ is allowed, e.g. when 'f'(x+y,x−y,x↑y) is applied to f(a,b,c), x↑y matches $((a+b)/2)^{((a-b)/2)}$.

Template ('POLY' ⋯) not only abbreviates a longer form ('+' ⋯) but is more powerful, e.g. template ('POLY' k x y) will match x+ky for any expression x and y (y≠0 and both free of k). An equivalent algebraic pattern will not.

4. **EXPRESSION TREES**. An *expression tree* for an expression e is a representation of e as a tree with ordered branches, with non-terminal nodes labeled with operators, and with terminal nodes labeled with integers and identifiers. The expression tree for an expression e is obtained by the following rules: (1) if e is an identifier or an integer, the expression tree of e is a single node labeled by e; otherwise, (2) e has the form (OP $a_1 \cdots a_n$) in which case its tree has the form:

where a_1', a_2',⋯, a_n' denote the trees for subexpressions $a_1,a_2,\cdots,a_n$ respectively. As usual we call OP the *root* of the tree. The left-to-right order of the subtrees given above corresponds to the parse tree produced by an input translator except that when OP is symmetric, $a_1' <\cdot a_2' <\cdot \cdots <\cdot a_n'$.

A *branch order*, a permutation of (1,2,⋯n), is defined for each OP and for each possible number n ($n \ge 1$) of arguments OP can take. Symmetric OPs always have branch order (1,2,⋯,n) after its arguments are ordered by $<\cdot$. An expression tree is traversed as follows. Initially visit the root of the tree. Then visit the remaining nodes of the tree using a *branch order traversal* defined as follows:

(1) visit the root of each subtree in branch order;
(2) traverse each subtree in branch order.

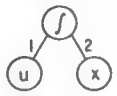
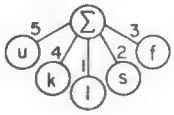
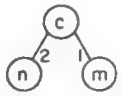
SYNTAX	INTERNAL FORM	EXPRESSION TREE
$\int_x u$	(∫ u x)	
$\sum_{k=i(s)}^{f} u$	(∑ u k i s f)	
$\binom{n}{m}$	(c n m)	
x^y	(↑ x y)	

Table 1.
Example internal representations and branch orders

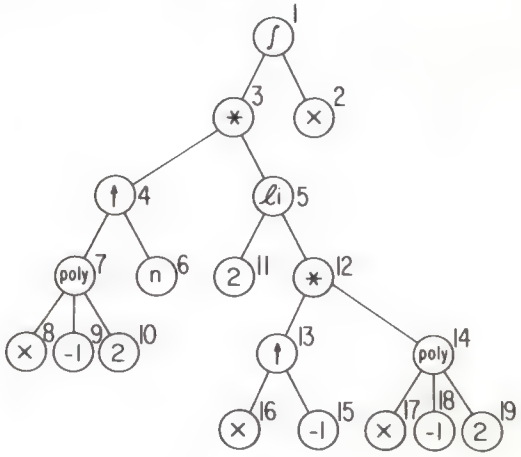
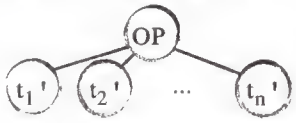


Figure 1.
Pattern Tree for Pattern (3).

Remarks. This non-standard definition of traversal provides a "breadth first" sweep across expression trees so as to quickly screen unsuitable subjects. Branch ordering plays a dual role of controlling how expressions are ordered during normalization and how expression trees are traversed during the matching process. In normalizing integral and sum patterns involving quotients, for example, a special branch order allows exponentials to be ordered first on their powers so that, for example, denominator terms having negative exponents will always appear before respective numerator terms. Also, binomial coefficients in a sum pattern can be ordered first on their lower argument which most often contains the summation index. During matching, branch order traversal forces the variable of integration for an integral pattern to be examined before the integrand so that ('POLY' ···) patterns can be used in matching the integrand.

5. **PATTERN TREES**. A *pattern tree* is used to describe a family of expressions matched by a given pattern. Pattern trees are similar to expression trees except that pattern trees have expressions at their terminal nodes. A pattern tree for a pattern p=(s,t,c) is obtained from template t by the following rules: (1) If t is an expression e, the pattern tree for p is a single node labeled by e; otherwise, (2) t has the form ('OP' $t_1 \cdots t_n$) in which case its tree has the form:



where t_1', t_2',···,t_n' denote the pattern trees for t_1, t_2,···,t_n respectively. For non-symmetric OPs, the left-to-right order of the subtrees above corresponds to the parse tree produced by an input translator. For symmetric OPs, subtrees are ordered by $<\cdot$ except perhaps for a *permutation set*, a set of two or more consecutive subtrees related by $=\cdot$. Permutation sets indicate subtrees that are ordered up to a (run time) permutation. Branch order and branch order traversal are similarly defined for pattern trees.

Remarks. By construction, a pattern p matches an expression e iff (1) the pattern tree for p is identical to the expression tree for e down to the terminal nodes of the pattern tree, and (2) there exists a match environment $(s \rightarrow a)_M$ such that terminal expressions $t_i(a)$ are algebraically equal to corresponding subexpressions e_i of e, i.e. $\mu(t_i(a)=e_i)$. We will call a pattern "fully compilable" if the relations $\{t_i(a)=e_i\}$ form a set of linear equations which can be solved for the match variables s_i.

6. **RECOGNITION TREES**. A *recognition tree* is derived from a pattern tree and provides a useful algorithm for recognizing a given pattern. A recognition tree is equivalent to a pattern tree except that terminal expressions of the pattern tree are replaced by *terminal predicates*. Let P denote a pattern tree for pattern p=(s,t,c) whose nodes are labeled 1,2,··· in order of visitation by branch order traversal (Figure 1). Let $I=(i_1,i_2,\cdots,i_n)$ denote the set of indices of the n terminal nodes t_i of P. In addition, let e_i denote the subexpressions of a subject e_1 matched by p whose corresponding subtree has its root at node i. The recognition tree for pattern p=(s,t,c) is derived from the pattern tree for p by replacing each terminal expression t_i by predicate p_i defined as follows. Beginning with c_0=match condition c, repeat the following for each i in I:

(1) Define $c_i' = (t_i - e_i = 0) \wedge c_{i-1}$ expressed in a conjunctive normal form.

(2) For each match variable s, declare s to be *bound at node* i if:

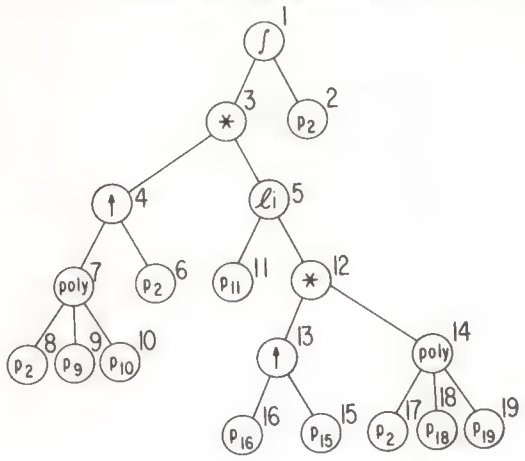
(a) s is defined by a set of linear equations appearing as conjuncts in c_i', and

(b) condition (a) holds at no time $k < i$.

(3) Define conjunctive normal forms c_i and p_i by the relation: $c_i' = p_i \wedge c_i$ such that p_i contains all conjuncts involving only match variables bound at node $k \leq i$. (Note: one or both of p_i and c_i may be **true**).

If a recognition tree is to properly represent a pattern, all match variables must be solvable by linear equations by the time the last node i is reached in branch order traversal. We say a pattern is *fully compilable* iff the above procedure ends with c_m=**true**.

Remarks. By construction, the process of deriving a recognition tree from a pattern tree distributes the terms of match condition c over the terminal nodes of the tree so that predicates may be applied at the earliest possible time. Note that the predicates p_i also include tests for multiple appearances of match variables in patterns (such as x and y in 'f'(x+y,x−y,x)). Note also that a match variable s bound at node j may be expressed in terms of $e_1,\cdots,e_j$. By construction, terminal predicate p_j involves only match variables s bound at node $i \leq j$ and thus also may be expressed solely in



p_2: **true**

p_6:	is e_6 integer $\wedge$ is $e_6 \leq -2$?	p_{15}:	is $e_{15}=-1$?
p_9:	is $e_9=-1$?	p_{16}:	is $e_{16}=e_2$?
p_{10}:	is $e_{10}=2$?	p_{18}:	is $e_{18}=-1$?
p_{11}:	is $e_{11}=2$?	p_{19}:	is $e_{19}=2$?

Figure 2.
Recognition Tree for Pattern (3).

terms of $e_1,\cdots,e_j$. Given a recognition tree it is now a simple matter to generate a recognizer.

7. **RECOGNIZERS**. A recognizer for pattern p will be represented by a sequence of *instructions* (Table 2) which systematically test that a given expression e is matched by pattern p. Assume the nodes of the recognition tree are labeled 1,2,⋯,m. Here we let $e_2,\cdots,e_m$ denote local storage cells which are available to hold subexpressions of subject e. Then a recognizer to test that expression $e=e_1$ is matched by pattern p=(s,('OP' ⋯),c) consists of a succession of tests and stores as output by the following procedure:

Initial Step: Output (EO e_1 (QUOTE =OP)) to test that e1 has main operator OP.

Steps i=1,⋯,m: For each node i of the recognition tree:

(1) if node i is labeled with a terminal predicate p, output predicate p expressed in terms of the values of $e_1,\cdots,e_i$;

(2) if the node is labeled with an operator OP' (≠ POLY) and dominates k nodes whose integer labels form the set $N_i=(n_i,n_i+1,\cdots,n_i+k-1)$:

(a) output (EL e_i =k) to test that e_i has k branches (if OP' is allowed a variable number of branches);

Stores:

(SC y x z)	**Store coefficients of x in y into z**
(ST y x)	**Store value of x into y**
(SA x y)	**Store arguments of x into y**
(SP x y)	**Store parts of x into y**

Tests:

(TI x)	**Is x an integer?**
(TN x)	**Is x an name?**
(EL x y)	**Is length(x)=y?**
(EO x y)	**Is main operator of x=y?**
(EQ x y)	**Is x=y?**
(NE x y)	**Is x≠y?**
(LE x y)	**Is x≤y?**
(LT x y)	**Is x<y?**
(GE x y)	**Is x≥y?**
(GT x y)	**Is x>y?**

Table 2.
Instruction Codes for Recognizer

```
(AND
  (EO e1 (QUOTE INT))
  (SA  e1 e3 e2)
  (EO e3 (QUOTE TIMES))
  (EL  e3 3)
  (SA  e3 e4 e5)
  (EO e4 (QUOTE EXPT))
  (EO e5 (QUOTE LI))
  (SA  e4 e7 e6)
  (EO e7 (QUOTE PLUS))
  (TI  e6)
  (LE e6 -2)
  (SC e7 e2 e8)
  (EL  e8 2)
  (SP  e8 e9 e10)
  (EQ e9 -1)
  (EQ e10 2)
  (SA  e5 e11 e12)
  (EQ e11 2)
  (EO e12 (QUOTE TIMES))
  (EL  e12 3)
  (SP  e12 e13 e14)
  (EO e13 (QUOTE EXPT))
  (EO e14 (QUOTE PLUS))
  (SA  e13 e16 e15)
  (EQ e15 -1)
  (EQ e16 e2)
  (SC e14 e2 e17)
  (EL  e17 2)
  (SP  e17 e18 e19)
  (EQ e18 -1)
  (EQ e19 2)
  (MV 3 e2 e6) )
```

Figure 3.
Recognizer for Pattern (6)

(b) for each $j \in N_i$, output (SA e_i ⋯) to store the k arguments of e_i into e_j;

(c) for each $j \in N_i$ such that node j is labeled by an operator OP_j, output (EO e_i (QUOTE =OP_j))) to test that expression e_j has main operator OP_j

(3) if the node is labeled with operator POLY and dominates n+2 nodes, say, $e_{i+1},e_{i+2},\cdots,e_{i+n+2}$:

(a) Output (SC e_i =e_{i+1} e_{i+1}) to deposit list of coefficients of powers of the index stored in e_{i+1} in e_i up to the leading coefficient into e_{i+1}.

(b) Output (EL e_{i+1} =n+1) to test that list e_{i+1} has length n+1;

(c) Output (SP e_{i+1} ⋯) to deposit the n+1 parts of e_{i+1} into $e_{i+2},\cdots,e_{i+n+2}$;

(d) For each j in (2,⋯,n+2): if node e_{i+j} has root OP_j, output (EO e_{i+j} (QUOTE =OP_j)) to test that the expression stored in e_j has main operator OP_j.

Two other types of nodes are briefly mentioned here. One is an ANCHOR node which signifies the need for a run time permutation to put terms into a fixed order. In this case, the compiler outputs complete information as to "what to look for"; this information is later used by the optimizer (see below). A second type of node is a PATTERN node which compiles into a call to a general simplifier. This node is used when the compiler cannot reduce a given pattern to a set of fully compilable patterns.

Recognizers are implemented by LISP AND-expressions (Figure 3) which are understood to have the following semantics. The arguments of an AND-expression are evaluated from left to right returning **false** (NIL in LISP) if one argument is found to be **false** or else the value of the the last argument if the list of arguments has been exhausted. Thus any value other than **false** is regarded as **true**. In particular, actions having value different from **false** can be executed in sequence for side effect without affecting its value. A recog-

nizer always ends with (MV i $u_1 \cdots u_n$) to return a denotation of a match environment $(s_0 \rightarrow i, s_1 \rightarrow u_1, \cdots, s_n \rightarrow u_n)$ where i denotes the "case-number" of the pattern.

8. **COMPILATION OF PATTERN SETS.** The process of compiling sets of patterns from a ruleset consists of 5 phases:

Phase 1: generation of patterns from ruleset expressions;
Phase 2: generation of recognizers from patterns;
Phase 3: merging of recognizers into a decision tree;
Phase 4: optimization of the decision tree;
Phase 5: compilation of the optimized tree.

Phase 1 consists of generating pattern triples (s,t,c) from expressions derived from given replacement rules of the ruleset. The general form of each replacement rule is

L=**if** C **then** R

where L is a expression denoting a template for a pattern, and where R denotes the replacement for L when the parameters of L satisfy condition C. Rules are generally split into two or more rules when the pattern derived from L is not fully compilable. This happens, for example, when condition C allows variables in L to range over identity elements of operators appearing in L. For example, the rule

$$\int_x x^n e^{ax} = x^n e^{ax}/a - (n/a)\int_x x^{n-1} e^{ax} \qquad (n \in (0,1,\ldots))$$

splits into three cases: n=0, n=1, and $n \in (2,\cdots)$. Rules also split when the prescribed range of variables allows terms to combine, e.g.

$$\int_t \cdots \sin(xt)\sin(yt) \quad (x,y \text{ arb})$$

splits into two cases: x=y and $x \neq y$.

Phases 2 and 3 are combined. Each pattern tree is successively transformed into a recognizer then merged into a single AND-OR expression representing a "decision tree" which becomes a recognizer for all patterns the given ruleset. A *decision tree* is a tree with ordered branches labeled with instructions and with terminal nodes labeled with a match environment. Branch nodes with two or more branches are implemented by the LISP expression (OR ···) which has semantics analagous to that of AND. Beginning with a partial decision tree represented by AND-OR expression d, a new recognizer implemented by (AND ···) is merged with d by repeatedly transforming an initial representation (OR (AND ···) d) by rewrite rule transformations:

(OR ··· (AND a ···) ··· (AND a ···) ···) →
  (OR ··· (AND a (OR (AND ···) (AND ···))) ···)

(op ··· (op ···) ···) → (op ··· ··· ···)
(op ··· a ··· a ···) → (op ··· a ··· ···) } op=AND,OR
(op a) → a

until no more changes result.

Phase 4 attempts to optimize the search over all patterns by performing transformations on the decision tree. One such optimization is to convert a branch into a *homogeneous* branch, that is, a n-way branch on the value of a single expression. Non-homogeneous nodes can frequently be "homogenized" by a process called "code-shifting". Let T(n) (resp. S(n)) denote an instruction (FN e_n ···) where FN is a test instruction (resp. store instruction). By *code-shifting* we mean a transformation of the form

(AND T(n) T(m) ··· ···) → (AND T(m) ··· T(n) ···)

which are logically equivalent and where T(n) is placed as far to the right as possible but to the left of any S(n). When the AND-expression contains no S(n) but ends with a branch node then T(n) is placed as far to the right as possible in each branch. In addition, compaction of code results from transforming expressions of the form

(OR ··· (AND ··· c··· ···)
    ··· (AND ··· c··· ···) ···)

with two common subsequences c··· into a logically equivalent form:

(OR ··· (AND c··· (OR (AND ··· ···)
                      (AND ··· ···))) ···)

Patterns with permutation sets give rise to *anchor nodes*, that is, a node which requires a dynamic search for the correct permutation of two or more arguments of a symmetric operator. Here it is useful to consider the nodes of the decision tree to be of two types. A *no-backup node* is such that if one of the successively examined branch predicates is **true**, then no other alternate branch predicates need be examined. A *backup node*, on the other hand, allows successive alternate predicates to be tested no matter how many previously examined branch predicates were **true**. Since the cost of searching over alternatives can be quite significant, the optimizer attempts to convert backup nodes into no-backup nodes in order to eliminate unneeded searching. A general strategy here is that of "node-splitting", that is, partitioning the set of branches into disjoint sets mutually requiring backup but where members of the respective sets can be handled by

```
(OR
 (AND
  (EO e1 (QUOTE INT))
  (SA  e1 e3 e2)
  (EO e3 (QUOTE TIMES))
  (EL  e3 3)
  (SA  e3 e4 e5)
  (EO e4 (QUOTE EXPT))
  (EO e5 (QUOTE LI))
  (SA  e4 e7 e6)
  (EO e7 (QUOTE PLUS))
  (TI  e6)
  (LE e6 -2)
  (SC e7 e2 e8)
  (EL  e8 2)
  (SP  e8 e9 e10)
  (EQ e9 -1)
  (SA  e5 e11 e12)
  (EQ e11 2)
  (OR
   (AND
    (EO e12 (QUOTE TIMES))
    (EL  e12 3)
    (SP  e12 e13 e14)
    (EO e13 (QUOTE EXPT))
    (EO e14 (QUOTE PLUS))
    (SA  e13 e16 e15)
    (EQ e15 -1)
    (EQ e16 e2)
    (SC e14 e2 e17)
    (EL  e17 2)
    (SP  e17 e18 e19)
    (EQ e18 -1)
    (EQ e19 2)
    (OR
     (AND
      (EQ e10 1)
      (MV 2 e2 e6))          (case (2))
     (AND
      (EQ e10 2)
      (MV 3 e2 e6) ) ) )     (case (3))
   (AND
    (EQ e12 e2)
    (EQ e10 1)
    (MV 1 e2 e6) ) ) ) )     (case (1))
```

Figure 4.
Optimized Decision Tree for 3 Patterns:
$\int_x u(n)$, $n \in (-2,-3,\cdots)$, where u(n)=
(1) $(x-1)^n LI(2,x)$; (2) $(x-1)^n LI(2,2-1/x)$;
(3) $(2x-1)^n LI(2,2-1/x)$ (same as pattern (6))

A Draft Description of the Group Theory Language Cayley

John J. Cannon

Department of Pure Mathematics
University of Sydney
Sydney

and

Forschungsinstitut für Mathematik
Eidgenössische Technische Hochschule
Zurich

§1. Introduction

In this paper we outline a language belonging to the domain-specific class of algebraic programming languages. The problem domain we are concerned with is that of the theory of discrete groups and related structures. In 1971, Neübuser in Aachen and Cannon in Sydney, commenced the development of a general purpose group theory system called GROUP, the great majority of which is coded in ANSI Standard FORTRAN. For a discussion of the group theory algorithms planned for the system see Cannon [1].

Two driver languages are planned, *Galois*, a language with explicit type declarations intended for batch processing, and *Cayley*, a language where types are determined at run time and hence suitable both for batch processing and interactive computing. An interpreter for Cayley has been implemented at Sydney for the CDC6000 and CYBER series machines. The interpreter is coded in ANSI Standard FORTRAN and experience indicates that the entire system (some 50,000 lines of FORTRAN at present) can be implemented on a new machine with less than 2 months programming effort.

§2. Design Philosophy

It should perhaps be noted from the outset that the design of Cayley has been influenced by the existence of programs implementing some very powerful algorithms (e.g. nilpotent quotient, Todd-Coxeter) and the desire to synthesise these into a unified system so that a number of these algorithms may be brought to bear on a problem within a single run. Again group structure algorithms such as the subgroup lattice algorithm generate as their output an enormous amount of information, most of which is irrelevant to the problem at hand. Thus our language is required to process and extract information from large and complex data structures.

Just as in ordinary programming languages, we wish to be able to program algorithms in a fairly natural way. There is the complication, however, that calculations may be performed in a number of different algebraic structures (e.g. groups, rings modules) and the precise specification of a structure (e.g. group of permutations of degree 6), need not be known until run time.

We define *type* to mean a set of values. Besides traditional types such as *integer* and *Boolean*, Cayley contains several classes of types, the two most important of which are the *class of algebraic structure types* and the *class of algebraic element types*. Examples from the former class of types are

permutation group of degree 6;
group of 4×4 matrices over GF(9);
finite field $GF(2^6)$;
ring of polynomials in x over field GF(2) modulo the polynomial x^7-1;
vector space of dimension 12 over GF(3).

Examples of types from the class of algebraic element types are

permutation of degree 6;
4×4 matrix over GF(9);
element of $GF(2^6)$;
polynomial over GF(2);
12-dimensional vector with coefficients from GF(3).

It can be seen that an algebraic structure type may consist of a single emement. There is a one-to-many correspondence between algebraic structure types and algebraic element types. In particular, it is necessary that an algebraic structure be defined before any of its elements.

The kinds of algebraic structure currently permitted in Cayley are group, ring, field, vector space and module. An algebraic structure is represented in the machine as a set of *attributes*. These attributes not only serve to define the particular algebraic structure but also save most of the information that has been computed about the structure. Thus attributes of a group include type, elements, conjugacy classes, subgroup lattice etc. Note that generally only a subset of the attributes of a structure are defined during a computation within that structure.

An important point to note concerning attributes is that their values need not belong to one of the standard Cayley types. An example of such an attribute is the subgroup lattice which is

no-backup. The need for no-backup nodes for sets of patterns with no permutation sets has yet to be demonstrated.

9. **TIMINGS**. A preliminary comparison was made of the performance of three pattern matchers in matching trigonometric integration formulas 336-357 from [6]. Both of the other two pattern matchers are implemented in SCRATCHPAD and consist of a set of compiled LISP functions which apply argument patterns to argument subjects in order to determine a match environment. The *general* pattern matcher is a modified version of the REDUCE pattern matcher. This pattern matcher finds all possible match environments for which the pattern structurally matches the subject then searches for one which satisfies the match condition. In the *revised* pattern matcher, the match condition is expressed as a logical-AND of univariate predicates, one for each match variable, and a multivariate predicate. Univariate predicates are carried along with the respective match variables into the matching process in order to immediately screen unsuitable subjects. The multivariate predicate is then applied to subjects as a final test for matching. In addition, all predicates are represented by "superprefix forms" which permit substitution without copying.

Preliminary tests show the new pattern matcher to be 6 to 200 times faster in recognition than the revised pattern matcher, 9 to 2000 times faster than the general pattern matcher, depending on the subject tested, and the order and style in which rules were issued. A detailed comparison of performance on an integral table from [2] and a table of combinatorial identities from [4] is planned for an expanded version of this paper to be published as an IBM Research Report.

10. **COMPARISON WITH MACSYMA; CONCLUSIONS**. The MACSYMA pattern matcher (described in [1]) is similar to that of SCRATCHPAD in several ways. Both produce compiled recognizers, avoid searching and backtracking during recognition, and provide for "semantic" as opposed to "syntactic" pattern matching (e.g. by regarding both $(x+2)(3x+1)$ and x^2+7x+2 as "quadratic in x"). Both use a rational canonical form to determine coefficients of powers of a variable when called for.

There are basic differences however. The MACSYMA compiler compiles single patterns not pattern sets as does the SCRATCHPAD compiler so that no advantage can be taken of information gained from previous attempts to match. Also, the MACSYMA compiler performs what might be called "algebraic pattern matching", i.e. matching by considering a "piece" of the pattern, deriving an expression from the subject which matches that piece, then recurring with a new pattern and new subject algebraically reduced by the matching parts. If when the reduced pattern becomes 0 the corresponding subject is 0, then the original pattern is said to match the subject. On the other hand, the SCRATCHPAD compiler might be said to perform "structural pattern matching". By static analysis at compile time, the compiler attempts to take full advantage of "advance" information about the internal representation of the class of expressions matched by a given pattern: the order of terms, the number of arguments the operator takes, the lead operator of subexpressions, etc. This information most often permits the determination of a match environment by solving equations. For example, in both systems the pattern 3^A+B^4 will match $1,3,3^z,w^4+1$ but will fail to match 10 which must be written in an unnormalized form for a match to be possible. Whereas MACSYMA would first attempt to apply the pattern 3^A to a subject by calling a match subroutine, SCRATCHPAD would split the general pattern into 10 cases, 9 patterns (1, 2, 3, 4, 3^A, 3^A+1, $1+B^4$, $3+B^4$, 3^A+B^4) which correspond to A and B=0,1, or not 0 or 1, and a tenth pattern(162) where A=B. In addition, SCRATCHPAD uses a special construct (POLY ···) in order to handle polynomial constructs efficiently. An extension of the existing system would allow a declaration that an expression e is "free of" a variable x. Whereas both systems would then allow the user to define a predicate "quadratic in x", the SCRATCHPAD version would be much simpler and usually more efficient (compare to [1, p. 323]):

```
(AND
 (SC e1 x e2)
 (EL e2 3)
 (SP e2 e3 e4 e5)
 (FO e3 x)
 (FO e4 x)          (FO means "free of")
 (FO e5 x)
 (MV 1 e3 e4 e5))
```

In addition, this facility could also be adapted and extended to compile pattern match constructs in a mode-based procedural language facility over a variety of data structures.

REFERENCES.

[1] Fateman, R. J., "The User-Level Semantic Matching Capability in MACSYMA", Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation, ACM, New York, 1971, pp. 311-323.

[2] Fox, John A. and Hearn, A. C., "Analytic Computation of Some Integrals in Fourth Order Quantum Electrodynamics", Journal of Computational Physics, Vol. 14, 1974, pp. 301-317.

[3] Griesmer, J. H., Jenks, R. D., and Yun, D. Y. Y., "SCRATCHPAD User's Manual", IBM Research Report RA 70, June 1975.

[4] Gould, H. W., Combinatorial Identities, A Standardized Set of Tables Listing 500 Binomial Coefficient Summations, Revised ed., Morgantown Printing and Binding Co., Morgantown, W. Va.

[5] Jenks, R. D., "The SCRATCHPAD Language", Proceedings of a Symposium on Very High Level Languages, SIGPLAN Notices, Vol. 9, No. 4, April, 1974. (Reprinted in SIGSAM Bulletin, Vol. 8, No. 2, May 1974).

[6] Peirce, B. O., "A Short Table of Integrals", Ginn and Company, New York, 1929.

more than just a set of subgroups. Attributes corresponding to non-standard Cayley types are created by functions which can only return a value for the attribute. Once created, such an attribute may be printed or its constituents, corresponding to standard Cayley types, may be referred to in ordinary statements.

Other important Cayley types are *set*, *sequence* and *mapping*. Because of the central role these concepts play in mathematics, any algebraic programming language without them would be seriously deficient. The elements of sets and sequences may be objects of any type. The domain and range of a mapping may be sets, sequences or algebraic structures. Special types associated with particular algebraic structures e.g. coset table with groups, are introduced as necessary.

A typical Cayley program will involve one or more variables of algebraic structure type. Since the computation of certain information about such a structure e.g. conjugacy classes, subgroups of a group, can be very expensive, it is saved (unless explicitly deleted by the user) as it will often be required further on in the computation. Thus as the computation proceeds more and more structural information is accumulated for each algebraic structure which is defined. Action routines are designed to check whether any of the saved information can be used to shorten their task.

In this paper we give a summary of the principal constructs in Cayley. A more complete description may be found in [2]. The form of presentation adopted here is modeled loosely on Wirth's admirable description of Pascal [5].

§3. Vocabulary, Identifiers, Integers and Constants

3.1. Vocabulary

We define the syntactic constructs of Cayley using an extended form of the Backus-Naur notation. Possible repetition of a construct is indicated by an asterisk (0 or more repetitions) or a circled plus sign (1 or more repetitions). If a sequence of constructs to be repeated consists of more than one element, it is enclosed by the meta-brackets { and }. If one of a number of different constructs <construct$_1$>, <construct$_2$>,...,<construct$_n$> must appear in a certain position, this is denoted by writing

$$\left\{ \begin{array}{l} \text{<construct}_1\text{>} \\ \text{<construct}_2\text{>} \\ \quad \cdot \cdot \cdot \\ \text{<construct}_n\text{>} \end{array} \right\}$$

We begin by defining the vocabulary or terminal symbols

<letter> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|
a|b|c|d|e|f|g|h|i|j|k|l|m|
N|O|P|Q|R|S|T|U|V|W|X|Y|Z|
n|o|p|q|r|s|t|u|v|w|x|y|z

<digit> ::= 0|1|2|3|4|5|6|7|8|9

A comment is

"<any sequence of symbols not containing ">"

and may be inserted between any two identifiers, numbers or special symbols. Normally, however, it will appear between statements. Special symbols are defined in [2].

3.2. Identifiers, Integers and Constants

<identifier> ::= <letter><letter or digit>*

<letter or digit> ::= <letter>|<digit>

Identifiers are used to denote constants, variables, attributes, procedures and functions.

<signed integer> ::= <integer>|<sign><integer>

<integer> ::= <digit>$^{\oplus}$

<sign> ::= +|-

<constant> ::= <signed integer>|'<character>$^{\oplus}$'|
true|false|null

§4. Types

In this section we describe the data types permitted in Cayley and indicate the possible values they can take. Because of the complexity of many of the types our description is necessarily informal.

4.1. Basic Types

integer

the values are the integers in a range defined by the implementation. The values are denoted by integers.

Boolean

(*true*,*false*)

string

the values are sequences of characters. Values are denoted by sequences of characters enclosed in primes.

4.2. Set, Sequence and Mapping Types

set

the values are collections of objects which themselves can be of any type. The elements of a set must all be exactly the same type and sets are unordered. The values are denoted by listing the elements within square brackets.

sequence

the values are ordered collections of objects which themselves can be of any type. The elements of a sequence must all be exactly the same type. Unlike in the case of sets, an

object may appear more than once in a sequence. The values are denoted by listing the elements within braces.

mapping

the values are mappings with domain and range of type algebraic structure, set or sequence. A mapping is defined by means of the define statement (section 9). Note that it is not necessary for a mapping to have the same type for its domain and range.

The distinction between *set* and *sequence* should be noted: sequences are ordered, sets are not; repetitions of elements are allowed in sequences but not in sets; the usual set theoretic operations are not applicable to sequences; the elements of a sequence may be indexed. Two standard functions are provided for manipulating sequences:

append(s,x) form the sequence consisting of the elements of sequence *s* followed by element *x*.

prune (s) form the sequence consisting of all but the last element of sequence *s*.

In addition, the functions *tseq(s)* and *tset(s)* are provided for changing the type of a set to that of sequence and the type of a sequence to that of set, respectively.

4.3. Class of Algebraic Structure Type

An object of type *algebraic structure* is any algebraic structure that can be defined in Cayley. Roughly speaking, specification of an algebraic structure type requries specification of how the elements are represented and the rules for performing whatever algebraic operations (e.g. multiplication, addition) that are appropriate. The totality of these types is called the *class of algebraic structure types*.

We now give an indication of the algebraic structure types that have been implemented or are planned for Cayley. We do not completely specify all the possible types because the range of possibilities is implementation dependent. Thus permutation group types, for example, might consist of permutation groups of degrees between 1 and 100,000 (recall that the permutation groups of a specific degree form a type). With the exception of integers, Gaussian integers and rationals, each sentence under the five following headings actually specifies a collection of types.

Rings: integers;
integers modulo m;
Gaussian integers;
polynomials with coefficients from some ring or field known to Cayley;
polynomials of degree n-1, with coefficients from a field known to Cayley, modulo an irreducible polynomial of degree n;
matrices over some ring or field known to Cayley.

Fields: rationals;
finite fields, $GF(p^n)$, p a prime;
cyclotomic number fields;
extensions of the rationals.

Groups: permutation groups;
matrix groups over some ring or field known to Cayley;
groups of geometric transformations (e.g. affine transformations);
groups given by defining relations and collection multiplication (Felsch [4]).

Vector Spaces: finite dimensional vector spaces over a field known to Cayley.

Modules: finitely generated modules over a ring known to Cayley.

Objects of type algebraic structure are defined either by means of the algebraic structure declaration statement (section 6), or within the scope of a structure S, by an expression defining a subobject of S.

We now list the attributes for type group. Appropriate attributes also exist for the other types of algebraic structure.

<attribute identifier> ::=
<group attribute identifier> |
<ring attribute identifier> |
<field attribute identifers> |
<vector space attribute identifier> |
<module attribute identifier>

<group attribute identifier> ::=
<group structure attribute identifier> |
<group property attribute identifier>

<group structure attribute identifier> ::=
name | type | size | degree | generators | relations | commutator power presentation | order | exponent | centralizer orders | class lengths | elements | classes | normal subgroup lattice | subgroup lattice | commutator subgroup | centre | Frattini subgroup | Fitting subgroup | derived series | lower central series | upper central series | Sylow subgroups | omega series | amego series | orbits | block systems | automorphism group | character table

<group property attribute identifier> ::=
cyclic | elementary | abelian | nilpotent | supersoluble | soluble | perfect | simple | transitive | primitive | regular | reducible | irreducible

4.4. Class of algebraic Element Types

An *algebraic element* type is the set of elements of any algebraic structure that can be

defined in Cayley. Consider, for example, the algebraic structure type comprising the set of permutation groups of degree 6. The elements of each such group are a distinct type, so that this single algebraic structure type gives rise to as many algebraic element types as there are permutation groups of degree 6.

The specification of an algebraic element type occurs when its parent algebraic structure is defined. So an algebraic element can be defined only if its parent structure already exists. Any ambiguity concerning the parent structure of an element must be resolved by specifying the structure to which the element belongs, as part of the expression defining the element.

4.5. Class of Coset Table Types

A *coset table* type comprises the set of coset tables corresponding to the subgroups of any group that can be defined in Cayley, with respect to a particular generating system for the group. Thus each distinct generating system for a fixed group will give a distinct coset table type. A coset table type is automatically specified whenever generators are defined for a group. As coset tables can only be constructed using one of a number of functions, the recognition of a coset table type presents no difficulties.

4.6. Class of Character Types

A *character* type consists of the ordinary (complex) characters of any group that can be defined in Cayley. Thus there is one character type corresponding to each group that can be defined in Cayley.

§5. Variables

<variable> ::= <identifier>

The type of most variables is determined by the structure and composition of the right hand side of the expressions defining them. Ambiguity can arise when distinguishing between elements and subsets of the current group, current ring, current field etc. Only in situations where ambiguity is likely to arise with these data types is it necessary for the user to declare the type of a variable by means of the of construction (see section 7.6).

§6. Algebraic Structure Declarations

Any computation in an algebraic structure must be performed within a previously declared structure of the appropriate type. Since there will usually be a number of different structures of similar types around, ambiguity can arise concerning which structure is to be used in a given operation. To avoid both this and the alternative of having to specify which structure is to be used as part of the statement, Cayley uses the concept of *current structure* i.e. current group, current ring, current field etc. To handle situations where one wishes to compute in several groups simultaneously without continually changing groups, the of construction is provided (see section 7.6).

<algebraic structure declaration> ::=
    <algebraic type class>:<variable>|
    <algebraic type class>:<variable>=<expression>

<algebraic type class> ::=
    group|ring|field|vector space|module

In the case

    <algebraic type class>:<variable>

where the variable is undefined, then the variable is generically typed according to the preceding special symbol. Subsequent attribute assignments will complete the type specification and define the structure. If a value already exists for the variable, then that variable should represent a previously defined structure, which now becomes the current one for that class, e.g. the current group.

The expression in the construction

    < algebraic type class>:<variable>=<expression>

must evaluate to an object of the kind described by the identifier. This construction types and defines the variable to be a particular algebraic structure. That structure then becomes the current one for its class.

§7. Expressions

7.1. Definition

<expression> ::=
    <variable>|<constant>|
    <unary operator><expression>|
    <expression><binary operator><expression>|
    <function designator>|
    (<expression>)|<expression>[<expression>]|
    <special identifier>.<expression>|
    <expression> of <variable>|
    <set >|<sequence>|<group>|
    <permutation>|<product of cycles>|<matrix>|
    <commutator>|<attribute definition>

The constructs appearing on the right hand side of this production are expanded in the following sections. Expressions are normally evaluated in the appropriate current algebraic structure, except as described in section 7.6.

Only five cases require separation according to precedence of operators. These are

    <special identifier>.<expression>
    <expression>[<expression>]
    <unary operator><expression> and <expression>
    <binary operator><expression>
    <expression> of <variable>

and are evaluated in order from the top to the bottom of this list. The precedence of the unary and binary operators is given in the following section.

7.2. Operators

<unary operator> ::= +|-|¬

<binary operator> ::=

<exponentiation operator>|<multiplying operator>|
<adding operator>|<relational operator>|
<negation operator>|<set or logical operator>

The operator precedences are given in the following table. Evaluation proceeds from the top to the bottom of the table and operators of equal precedence are processed from left to right.

number	operator class	operators
1	exponentiation	↑,↑-
2	multiplication	*, /, mod
3	adding	+, -, unary-
4	relational	=,≠,<,≤,≥,>,⊂,⊆,⊇,⊃,in
5	negation	¬
6	set or logical	∩,∧
7	set or logical	∪,∨

With appropriate operands, the operators in this table have the usual meanings. For example, the two operands for the operator + may both be integers, ring elements, field elements, vectors or module elements. In some cases, however, the meaning of operators has been extended. The most important extensions are the following: If both the operands for ↑ are group elements then the operation is conjugation. If both the operands for / are algebraic structures then the result is a quotient structure. If the operands for + are group elements, then the commutator of the two elements is formed. If the two operands for - are both sets then the operation is set difference. The use of ¬ with a set operand creates the set complement. For operators * and /, either or both operands may be sets of algebraic elements, while the first operand for ↑ may also be a set of elements. Tables 1-6 give detailed semantics for the operators.

7.3. Function Designators

A function designator specifies the invocation of a function or a mapping.

<function designator> ::=

<function identifier>(<actual argument list>)|
<function identifier>(<actual argument list>)
<parameter list>|
<mapping identifier>(<argument list>)|
<read function designator>

<function identifier> ::= <identifier>

<actual argument list> ::=

<actual argument>{,<actual argument>}*

<actual argument> ::=

<expression>|<function identifier>|
<mapping identifier>|<procedure identifier>

<parameter list> ::= <parameter>{,<parameter>}*

<parameter> ::= <parameter identifier>|
<parameter identifier>=<expression>|
<parameter identifier>:<identifier>

<parameter identifier> ::= <identier>

<mapping identifier> ::= <identifer>

<procedure identifier> ::= <identifier>

From the syntactic viewpoint mappings are used in exactly the same way as functions. Cayley contains a special kind of function called a *parametrised function*, that is, a function which in addition to arguments, has *parameters*. We distinguish between arguments which correspond to the operands on which the function operates, and parameters which control the way in which the function operates. A common parameter is a time limit for the execution of the function. Each parametrised function is designed so that if the user does not specify values for any or all of the parameters, then default values are assigned. However, the existence of parameters allows the experienced user to solve much harder problems. It is important to note that parametrised functions may not be embedded in expressions. For those parametrized functions for which this may be desirable, non-parametrised versions are provided.

Functions must always be called with the same number of arguments. Every function must have at least one argument. A parametrised function may be called with none or a subset of its parameters assigned.

A particular function argument need not be restricted to a single type. For example, the Todd-Coxeter function has two arguments corresponding to a group and subgroup. Each of the two arguments may either be an expression defining a group or an expression defining a set of words.

Algebraic structures current at the time a function is invoked will be current upon return from the function.

The read function will be described in section 11.

Examples: In the following examples, G is a group; H is a subgroup of G; a and b are elements of G;

TABLE 1.

Exponentiation Operator

<exponentiation operator> :: = ↑

operation	type of first operand	type of second operand	type of result
exponentiation	integer	integer	integer
exponentiation	ring element	integer	ring element
exponentiation	field element	signed integer	field element
exponentiation	group element	signed integer	group element
conjugation	group element	group element	group element
exponentiation	set of integers	integer	set of integers
exponentiation	set of ring elements	integer	set of ring elements
exponentiation	set of field elements	signed integer	set of field elements
exponentiation	set of group elements	signed integer	set of group elements
conjugation	set of group elements	group element	set of group elements
action	integer	permutation	integer
action	vector	matrix	vector

Notes

(i) If a and b are group elements, then a↑b is the element $b^{-1}ab$.

(ii) (a) Suppose G is a permutation group acting on the set Ω. If x ε G and i ε Ω, then i↑x denotes the element of Ω to which i is mapped under the action of permutation x.

(b) Suppose G is a matrix group acting on the vector space V. If x ε G and v ε V, then v↑x denotes the vector-matrix product, i.e. the image of v under the action of x.

Both (a) and (b) can be generalized to the case where one or both operands are sets of elements of the appropriate types. Thus if I is a subset of Ω and X is a subset of elements of permutation group G then I↑X is the orbit of I under the action of X.

TABLE 2.

Multiplication Operators

<multiplication operator> ::= *|/|mod

operator	operation	type of operands	type of result
*	multiplication	integer	integer
*	multiplication	algebraic element	algebraic element
*	multiplication	set of algebraic elements	set of algebraic elements
/	division	integer	integer
/	division	field element	field element
/	$a/b = ab^{-1}$	a,b algebraic elements	algebraic element
/	$R/S = \{ab^{-1} \mid a \varepsilon R, b \varepsilon S\}$	R,S sets of algebraic elements	set of algebraic elements
/	quotient structure	algebraic structure	algebraic structure
mod	remainder modulo an integer	integer	integer

Notes

(i) Multiplication and division are only permitted between operands that make sense algebraically e.g., multiplication between two elements of the same group; multiplication between an element of a ring R and an element of a module over R.

(ii) Multiplication of a set S and an element a is permitted. Thus $a*S = \{ax \mid x \varepsilon S\}$ and $S*a = \{xa \mid x \varepsilon S\}$.

(iii) The product of two sets of elements R and S is defined by $R*S = \{xy \mid x \varepsilon R, y \varepsilon S\}$.

(iv) $m/n = \lfloor \frac{m}{n} \rfloor$.

(continued)

(v) The operator / may only be used in algebraic structures where each element has a multiplicative inverse, e.g. in groups, fields and division rings.

(vi) The constructions a/S and S/a are permitted. Thus $a/S = \{ax^{-1} | x \in S\}$ and $S/a = \{xa^{-1} | x \in S\}$.

(vii) n mod m has its conventional mathematical meaning i.e. the integer r, $0 \leq r \leq m - 1$ such that $r = n + qm$.

TABLE 3.

Addition Operators

<addition operator> ::= +|-

operator	operation	type of operands	type of result
+	addition	integer	integer
+	addition	additive element	additive element
+	commutator	group element	group element
+	commutator	set of group elements	set of group elements
-	subtraction	integer	integer
-	subtraction	additive element	additive element
-	set difference	set	set

Notes

(i) Any operation applicable to a set is automatically applicable to the set of elements comprising an algebraic structure. For example, all operations applicable to sets can be applied to subgroups. However, the result will not always be a group type.

(ii) The first and second operands must be of the same type.

(iii) An additive element is any element of a ring, field, vector space or module.

(iv) When used as a unary operator, - denotes sign inversion in the case of an additive element, and multiplicative inverse in the case of a group element, while + denotes the identity operation.

(v) The expression S-a, S a set and a an element, denotes the removal of a from the set S.

TABLE 4.

Relational Operators

<relational operator> ::= =|≠|<|≤|≥|>|⊂|⊆|⊇|⊃|in

operator	type of operands	type of result
= ≠	any type, except algebraic structure and coset table types	Boolean
< > ≤ ≥	integer	Boolean
⊂ ⊃ ⊆ ⊇	set, algebraic structure	Boolean
in	second operand: any set type including algebraic structure; first operand: any object of the same type as the elements of the second operand set.	Boolean

TABLE 5.

Negation Operator

<negation operator> :: = ¬

operator	operation	type of operand	type of result
¬	negation	Boolean	Boolean
¬	complementation	subset	subset

Notes

(i) The operator ¬ applied to a subset, denotes the set theoretic complement

TABLE 6.

Set and Logical Operators

<set or logical operator> :: = ∩|∧|∪|∨

operator	operation	type of operands	type of result
∨	logical "or"	Boolean	Boolean
∧	logical "and"	Boolean	Boolean
∪	union	set	set
∪	union	algebraic structure	algebraic structure
∩	intersection	set	set
∩	intersection	algebraic structure	algebraic structure

Notes

(i) For unions and intersections of algebraic structures, the two operand structures must be represented in exactly the same way e.g., two permutation groups of degree 12.

(ii) The union of two algebraic structures is defined to be the smallest structure of the same type containing them.

(iii) The construction S∪a, S a set and a an element, is permitted. It has the effect of adding a to S if it was not already in S.

R and *S* are sets of words in the generators of *G*; *phi* is a mapping from group *K* to group *G*, *x* is an element of *K*.

```
order(<a*b,b↑2>)
normalizer(stabilizer([2,3,4]))
elements(H) al:Schreier,base = {2,3,5},tl = 10
Todd Coxeter(G,H)
Tod Coxeter(R,S)max=10000,min=100,tl=10
phi(x)
```

The *elements* function example has parameters which specify that the Shreier permutation group algorithm is to be used with initial base {2,3,5} and time limit 10 seconds.

The second *Todd Coxeter* example specifies that the enumeration is to use a maximum table size of 10000,with a minimum lookahead recovery of 100 cosets and a time limit of 10 seconds.

7.4. Indexing Expressions

The components of vectors and sequences can be accessed by means of the construction

<expression>[<expression>]

where the first expression must evaluate to a vector or sequence, and the second to a non-negative integer not exceeding the number of components.

Examples: Suppose *L* is a sequence, *M* is a sequence of sequences, *v* is a vector, *i* and *j* are integers.

v[*i*]

L[5]

L[2*i*+*j*]

derived series (*G*) [3]

M[3][5]

7.5. Special Identifiers

The construction

<special identifier>.<expression>

is used for accessing components of structures which do not correspond to any standard Cayley type. Thus, for example, if the subgroup lattice of a group has been computed (and is present as a group attribute), the expression S.3 has as its value the third subgroup in the lattice. While the subgroup lattice is not a standard Cayley type, its component subgroups, and the expression S.i, i an integer, enables individual subgroups in the lattice to be accessed.

This construction is also used to refer to the components of certain attributes that are standard Cayley types. For example, if the list of elements of a group has been printed the user can refer to a particular element using a pre-defined identifier which is printed alongside the element. Thus, in the case of groups, the routines responsible for printing classes, elements, normal subgroups, subgroups, characters etc. print an identifier of this kind alongside each component.

<special identifier> ::= C|E|N|S|X

The special identifiers designate class, element, normal subgroup, subgroup and irreducible character, respectively.

Examples:

E.5*a*

H∩*S*.7

normalizer(*S*.12)

7.6. Designating the Parent Structure

As noted earlier, calculation normally proceeds in the current group, the current ring etc. If it is desired to work in a structure that is not current, without making it current, then the of construction is used. It has the form

<expression> of <variable>

where the variable must denote some existing algebraic structure and the expression must define a legal entity in that structure. In particular, this enables information from other structures to be used within the current one. Further, it provides a means of resolving ambiguities, where it is not clear, for example, whether a "constant" element belongs to the current ring or group.

Examples: *G* is the current group; *H* is another group having the same kind of representation as *G*; *a* and *b* are elements of *H*; *c* is an element of *G*

*(a*b of H)*c↑-1*

centralizer(G,a of H) ∩ normalizer(G,c)

§7.7. Sets, Sequences and Groups

<set> ::= [<expression>{,<expression>}*]|
[<element domain>{,<element domain>}*:
<expression>]|
[ ]

<sequence> ::= {<expression>{,<expression>}*}|{ }

<group> ::=<<expression>{,<expression>}*>|< >

<element domain> ::= <free variable>ε<expression>

<free variable> ::= <identifier>

A set may be defined either by enclosing the lsit of its elements within square brackets or by listing the properties its elements must satisfy. A set must have as its elements entities of any type but every element of a particular set must have the same type. Thus the values of expressions occurring in the list form of set definition must all be of the same type. The expressions occurring

in the element domains for the property form of set definition must define sets of the same type. The expression following the colan must be a Boolean expression having as variables the free variables occurring before the colan. If there are n free variables occurring in the set definition then the resulting set will be a set of sequences of length n (i.e. a set of n-tuples).

The expressions occurring in the definition of sequence must all define entities of the same type.

The expressions occurring in the definition of group must either define elements, sets of elements, or subgroups in the same group.

Examples:

[3,5,7,8]

[(1),(1,2),(1,3),(2,3),(1,2,3)]

[a↑2, b↑3,(a*b)↑2]

[x∈K:order(x)=2 ∧ x in normalizer(H)]

{1,2,4,13,11}

<a,b↑2>

<(1,2),(2,3,4)>

{<(1,7,5,6,3)(4,9,10),(2,8)>,<(2,8),(4,10,9)>,
<(2,8)>}

7.8. Permutations, Products of Cycles and Matrices

We now give the productions which allow the definition of permutations and matrices.

<permutation> ::= /<integer>{,<integer>}*/

<product of cycles> ::= {<cycle>}⊕

<cycle> ::= (<integer>{,<integer>}⊕)

<matrix> ::= (<expression>{,<expression>}⊕)

A permutation may either be written in image form or as a product of cycles. If the permutation group is of degree n, then the permutations are assumed to act on the set {1,2,...,n}. A permutation in cycle form must have cycles of length one omitted. The identity is written as (1).

Matrix entries are written row-by-row. Expressions defining matrix entries must have type ring element or field element and, for a particular matrix, must all belong to the same ring or field. Since all matrices that appear in a Cayley program must belong to a previously defined group or ring, the ring or field to which the entries of a given matrix belong is ordinarily found by examination of the attributes of the current group or ring. If the matrix belongs to a structure that is not current, or if ambiguity is possible, then the of construction must be used.

Examples: The permutation $\begin{pmatrix}1&2&3&4&5&6\\2&3&1&4&6&5\end{pmatrix}$ may be written either in image form or as a product of cycles as follows:

/2,3,1,4,6,5/

(1,2,3)(5,6)

The matrix $\begin{pmatrix}1&0\\2&-1\end{pmatrix}$ is written as

(1,0,2,-1)

The matrix $\begin{pmatrix}1+x&0\\x&1\end{pmatrix}$, x an element of GF(4), is written as

(1+x,0,x,1)

7.9. Commutators

<commutator> ::= (<expression>{,<expression>}⊕) |
<expression>+<expression>

The construct commutator is included so as to provide the usual notation for the commutator of a pair of group elements: $(a,b) = a^{-1}b^{-1}ab$. The expressions in the definition of commutator must either define elements or sets of elements in the same group. If brackets are omitted, left norming is assumed.

7.10. Special Attribute Definition Forms

In order to allow the algebraist to define structures in familiar ways a number of special expression forms are provided that may only be used in the context of an attribute assignment. In particular, special forms are provided when specifying generators and defining relations for groups.

<attribute definition> ::= <generators expression> |
<relations expression>

<generators expression> ::= <variable>
{=<expression>}{,<variable>{=<expression>}}*

<relations expression> ::= <relation expression>
{,<relation expression>}*

<relation expression> ::=
<expression>{=<expression>}* |
<variable>:=<expression>

Each occurrence of expression in the definition of generators expression must have as its value an element of the group. The generators expression allows the user to

(i) define generator names
(ii) define generators
(iii) define generators with specified names.

The relations expression construction is designed to allow the user great flexibility when inputting relations. Our objective is to closely approximate the familiar forms of writing relations. Thus we might write

$$a^2, b^3, (ab)^3$$

or

$$a^2=b^3=(ab)^4=1$$

or

$$a^2=b^3=(ab)^4$$

Further, we may wish to use abbreviations for long words that frequently occur in a relation set. All of these forms are catered for in the relations expression.

Recall that by a word in a group we mean the formal product of generator variables and their inverses. Each of the expressions occurring in the definition of relation expression must have as its value a word in the current group. The symbol:= in this context indicates that the variable on the left hand side is to be used as an abbreviation for the word on the right hand side during input. Every time such a variable is encountered in the input it is replaced by the word on the right hand side of its defining construction. The constructions defining these abbreviations do not appear in the relations stored in the machine. In order to avoid circularity, the variable occurring on the left hand side of := may not appear in the expression on the right hand side either explicitly or implicitly.

If generators have not been defined for the group at the time a relations expression is executed, then the generators occurring in the relations expression will be assigned to the group.

The trivial word (identity) is denoted either by the digit 1 or (1).

Examples:

generator expressions:

$a, b, x1, x2$

$a=(1,2)$, $b=(2,3,4)$

$a=(1,0,0,1)$, $b = (2,1,0,1)$

$a1=x\uparrow 2*y$, $a2=y\uparrow 2$

relations expressions:

$a\uparrow 2, b\uparrow 3, (a*b)\uparrow 4$

$a\uparrow 2=b\uparrow 3=(a*b)\uparrow 4=1$

$a\uparrow 2=b\uparrow 4=(a*b)\uparrow 11=1, x:=a*b, y:=a*b\uparrow 2, z:=a*b\uparrow 3,$

$y\uparrow 3=(x*y*z)\uparrow 3=x\uparrow 2*z\uparrow 2*x*z*y*x*z=1$

§8. Statements

<statement list> ::= <statement >{;<statement>}*

<statement> ::=

<simple statement>|<compound statement>

<simple statement> ::=

<assignment statement>|
<algebraic structure declaration>|
<mapping definition>|
<procedure statement>|
<input/output statement>|
<delete statement>|
<null statement>|
loop|break|result|results|return|quit

<null statement> ::=

Algebraic structure declarationsare discussed in section 6, mapping definitions in section 9 and input/output statements in section 10.

8.1. Simple Statements

8.1.1. Assignment Statements

An assignment statement either redefines the value of a variable or an attribute.

<assignment statement> ::=

<variable>=<expression>|
<variable>:=<expression>|
<variable list>=<function designator>|
<identifier>{[<expression>]}⊕=<expression>|
<identifier>{[<expression>]}⊕:=<expression>|
<attribute identifier>:<expression> |
<attribute identifier>:<attribute definition>|
<function designator>|
<function name>=<expression>

<variable list> ::= <variable>{,<variable>}*

An important point to note about the assigment is the alternative forms

<variable>=<expression>

or

<variable>:=<expression>

and

<identifier>{[<expression>]}⊕ =<expression>

or

<identifier>{[<expression>]}⊕:=<expression>

The form denoted by the symbol = corresponds to the traditional assignment in which the expression on the right hand side is evaluated and this value is assigned to the left hand side. The form denoted by the symbol := differs in its effect from the first form only when the right hand expression is a product of generator symbols and their inverses for the current group. In this case the unevaluated expression (word) is assigned to the left hand side. If no values exist for the generators of the group, then the unevaluated expression is assigned irrespective as to whether = or := was used. The existence of these two forms of

assignment make it possible for the user to work with elements both as abstract words and as elements of the representation.

Example:

group: S4; degree:4; generators:a=(1,2),b=(2,3,4);
*x=b↑2;y=a*b;u:=b↑2;v:=a*b;*

Here x and y have as their values the permutations (2,4,3) and (1,3,4,2), respectively, while u and v have as their values the words $b\uparrow 2$ and $a*b$, respectively.

The <identifier>{[<expression>]}$^{\oplus}$ construction on the left hand side of an assignment allows the assignment of values to the components of vectors and sequences.

Examples:

v[4]= -3

chain[3] = <*a*,*a***b*>

S[2][5] = *centralizer(H)*

S[3] = {*K*,*L*,*M*}

A small number of built-in functions are allowed to return multiple values. In order to provide for this we allow multiple assignments in the case where the right hand side is a function designator. If the function on the right hand side of an assignment statement potentially returns n values, but only m < n variables are listed on the left hand side, then the first m values only of the function are assigned.

Examples:

index, ct = Todd Coxeter(G,H)

*index = Todd Coxeter([a↑2,b↑3,(a*b)↑5][a↑-1*b↑2])*

The names of attributes act in effect as built-in identifiers and values can be assigned to them by using : in place of = or := .

Example:

group:A5;degree:5;generators:a=(1,2)(4,5),b=(1,3,4);
*relations:a↑2=b↑3=(a*b)↑5=1;order:60;*

Note that the values of a and b in the *generators* assignment must satisfy the relations.

Most group attributes are constructed by built-in functions. If the designator of a function which can be recognised to calculate some attribute appears by itself, then the value of that function is assigned to that attribute. Thus, for example, the statement

elements(A5)

may be considered as abbreviation for

elements:elements(A5)

Example: Suppose we want the elements, classes and subgroups of A5.

group:A5;degree:5;generators:a=(1,2)(4,5),b=(1,3,4);
elements(A5);classes(A5);subgroup lattice(A5);

8.1.2. Procedure Statements

<procedure statement> ::=

<procedure identifier>|
<procedure identifier>(<actual argument list>)

<procedure identifier> ::= <identifier>

All structures current when a procedure is involved will be current on return.

8.1.3. Delete Statement

Since the value of a variable or attribute may occupy a significant amount of space it is often necessary to delete the values of variables and attributes which are no longer required.

<delete statement> ::= delete <delete list>

<delete list> ::=

$$\left\{\begin{matrix}\text{<variable>}\\ \text{<attribute identifier>}\end{matrix}\right\}\left\{,\left\{\begin{matrix}\text{<variable>}\\ \text{<attribute identifier>}\end{matrix}\right\}\right\}^{*}$$

8.2. Structured Statements

<structured statement> ::= <compound statment>|
<conditional statement>|
<repetitive statement>

8.2.1. Compound Statements

<compound statement> ::= begin<statement list>end

A compound statement thus comprises a set of statements which have to be executed in sequence.

8.2.2. Conditional Statements

<conditional statement> ::=
<if statement>|<cases statement>

<if statement>::=
if <expression> then <statement list>end|
if <expression> then <statement list>else
<statement list>end

<cases statement> ::= cases$\left\{\text{<case>}\right\}^{\oplus}$end

<case> ::= case<expression>$\left\{,\text{<expression>}\right\}^{*}$:

<statement list>{break | <null statement>}|
default<statement list>

The expression following if in the if statement must be Boolean.

Examples:

```
if abelian(S.i) ∧ order(S.i)=8 then print S.i
                                    end

if order(H)=120 then K=H else K=<H,a*b> end
```

The cases statement is somewhat different from the traditional case statement in that the separate cases correspond to independent Boolean expressions. We have choosen this form since it seems more useful in the context of algebraic programs.

The expressions following each occurrence of case must be Boolean. Normally after executing a particular case, you fall through to the next unless the case is terminated by a break statement. This causes control to be transferred to the first statement following the cases statement. The default gets done if none of the other cases are satisfied. It may only appear once and must not be followed by cases. Note that is possible to have more than one Boolean expression "labelling" the same case.

Example: Suppose we wish to print which of a number of properties is possessed by a subgroup H. The properties are not mutually exclusive, so that several can be true simultaneously. Built-in Boolean-valued functions are available to test for each property.

```
cases
   case abelian (H)  :  print 'abelian' break;
   case nilpotent(H) :  print 'nilpotent' break;
   case soluble (H)  :  print 'soluble' break;
   case perfect (H)  :  print 'simple' break;
   case simple (H)   :  print 'perfect';
end
```

8.2.3. Repetitive Statements

<repetitive statement> ::=
   <while statement>|<repeat statement>|
   <for statement>

8.2.3.1. While Statements

<while statement> ::=
   while<expression>do<statement list>end

The expression following while must be of type Boolean and the statement is repeatedly executed until the value of the expression becomes *false*. If its value is *false* at the start of execution, the statement will not be executed at all.

Example:

```
while index≠28 ∧ i<4 do
      i=i+1
      index=Todd Coxeter(RUW[i],S);
end;
```

8.2.3.2. Repeat Statements

<repeat statement> ::=
  repeat<statement list>until<expression>

The expression following until must be of type Boolean. The statements between repeat and until are executed repeatedly until the execution becomes *true*. Notwithstanding the initial value of the expression, the statement list is always executed at least once.

Example:

```
repeat  K=H;
        H=normalizer(K);
until   H=K;
```

8.2.3.3. For Statements

<for statement> ::=
   for <control variable>=<for list>do
                        <statement list> end|
   for each <control variable>in<expression>do
                        <statement list> end|
   for each <attribute element><control variable>do
                        <statement list> end

<control variable> :: = <identifier>

<for list> ::= <initial value>to<final value>|
               <initial value>to<final value> by
                                    <increment>

<initial value> ::= <expression>

<final value> ::= <expression>

<increment> ::= <expression>

<attribute element> ::= generator|element|class|
                        subgroup|subgroup class|
                        subgroup rep|
                        normal subgroup|character

The initial value, final value and increment must all be integer valued expressions and their values are computed only upon entering the loop. Thus it is impossible to redefine the range or increment from within the loop.

In the first form of the for each construction, the expression must define a set or sequence and the control variable becomes an element of that set or sequence. The second form of for each, e.g. for each subgroup, is provided to allow repeated

execution over the components of attribute structures that are very frequently used.

Two special statements, loop and break, are provided for modifying the flow of control in the body of the loop. The presence of loop anywhere in the statement list, causes the commencement of the next iteration. On the other hand, break transfers control to the statement following the loop construction, i.e. right out of the loop.

Examples:

```
for i=1000 to 10000 do if prime (i)
        then print i end end

for each x in H do
for each y in K do
        comm=comm ∪ (x,y);
end;
end;

for each element a do
        H=H ∩ H↑a;
end;

for each class C do
for each x in C do
        if x in H then K=K∪x end;
end;
end;

for each subgroup rep S do

 if order(S)=32∧cyclic(centre(S)) then print S
end;
end;
```

§9. Mapping Definitions

Because of the central role played by mappings in algebraic structures we have included them as a class of types in their own right. While mappings could be achieved through the use of Cayley functions, their status as an independent data type, together with the presence of a mapping definition statement, enables them to be implemented very efficiently and provides a notation similar to that used by algebraists. In this draft we have restricted the possibilities for defining mappings to those that occur most commonly in group theory.

A mapping is used in the same way as a function reference.

```
<mapping definition> ::=
  define<map identifier>:<variable>↠<variable> by
        <expression>↠<expression>{,<expression>↠
                                  <expression>}*|

  define<map identifier>:<variable>↠<variable>by
    <map identifier>(<variable list>)=<expression>|

define<map identifier>:<variable>↠<variable>by
                              <generator images>|

  define{isomorphism / homomorphism / inclusion}<map identifier>:
    <group identifier>↠<group identifier>|

  define homomorphism<map identifier>:
    <group identifier>↠<group identifier>
     by <group identifier>/<group identifier>

  define{automorphism / endomorphism}<map identifier> of
        <group identifier> as <variable> of
                              <group identifier>

<map identifier> ::= <identifier>

<group identifier> ::= <identifier>

<subgroup identifier> ::= <identifier>

<generator images> ::=
   <generator image>{,<generator image>}*

<generator image> ::= <map identifier>(<variable>)=
                              <expression>
```

In the first two possibilities for mapping definitions, the variables occurring in <variable>↠<variable> preceding the by can be any algebraic structure, set or sequence type. The first form of define specifies a mapping by listing the image of each element in the domain. Clearly this is only practical where very small sets are involved.

Example

```
Suppose S = [1,(1,2),(1,3),(2,3),(1,2,3),(1,3,2)]
        R = [+1,-1]

 define  parity : S → R by

1↠+1,(1,2)↠-1,(1,3)↠-1,(2,3)↠-1,(1,2,3)↠+1,(1,2,3)↠+1
```

The second form of define describes a mapping between two objects by giving a rule for the calculation of the image of any element.

Examples:

(i) Suppose G is generated by elements a and b while $H \cong C_p \times C_q$ is the commutator factor group of G. We may define the homomorphism from G to H as follows: (Assume *exsum* is a function returning the sum of the exponents of a designated generator in a word).

```
define  abel: G → H by
        abel(x)=a↑mod(exsum(a,x),p)*b↑mod
                                (exsum(b,x),q)
```

(ii) Suppose y is a previously defined element of G. Then we can define the inner automorphism induced by y as follows:

```
define  alpha: G → G by alpha(x)=x↑y
```

(iii) Standard homomorphisms of a group can be easily created:

```
define  hom1: G → K by
        hom1 = permutation representation (H)
```

This defines *hom1* to be the permutation representation of G on the cosets of subgroup H.

```
define  hom2: G → K by hom2=transfer(H)
```

This defines *hom2* to be the transfer of G into H.

```
define  hom3: G → K by hom3=blocks image(blksys)
```

This defines *hom3* to be the homomorphism obtained by allowing permutation group G to act on the block system *blksys*.

The third case allows a mapping to be specified by giving the images of generators for the domain structure. Thus the domain must be a group, ring,vector space or module for which generators (basis elements) are available. Except in the case in which the generator images are elements of the domain, the image structure must already be set up. In the former case, the image structure will be automatically set up by the define statement. Note that the generators' images must satisfy the condition that the resulting mapping be a homomorphism.

Example.

Assume that group G is generated by a and b while group H is generated by x and y

```
define  phi: G → H by
        phi(a)=x*y,
        phi(b)=y^-1

define  rho: G → K by
        rho(a)=a↑-1,
        rho(b)=b↑a
```

Apart from defining *rho*, the group K will be created by this statement.

The fourth case of define allows the definition of three standard mappings between two groups. In the case of isomorphism and homomorphism, an onto mapping is constructed if one exists. Otherwise, the execution of the statement fails. In the case of inclusion an isomorphism is constructed (if one exists) between the group that is the domain and a subgroup of the range.

The fifth case creates the canonical homomorphism between a group and one of its factor groups. The subgroup under the slash must be normal.

The sixth case allows the constructuion of mappings corresponding to previously computed automorphisms or endomorphisms of a group.

Examples:

```
define isomorphism  f: G → H
define homomorphism h: G → K
define homomorphism hom: G → IMAGE by G/N
define automorphism a of G as x of AUTG
```

In the last example x is a previously defined element of *AUTG* which is the automorphism group of G.

§10. Procedure and Function Declarations

```
<procedure declaration> ::=
        <procedure heading><statement part>

<function declaration> ::=
        <function heading><statement part>

<procedure heading> ::= procedure<identifier>; |
                        procedure<identifier>
            (<formal parameter list>);

<function heading> ::= function<identifier>
            (<formal parameter list>);

<formal parameter list> ::= <identifier>
                                {,<identifier>}*

<statement part> ::= <compound statement>
```

The identifiers appearing in the formal parameter list and the statement part are local to the procedure or function declaration and hence cannot be accessed from other parts of the program. Similarly, identifiers occurring outside a procedure or function declaration cannot be accessed from within it. Procedures and functions may be called recursive.

Functions may return one or more values. These values are returned by a statement of the form

```
{result}
{results}  <expression>{,<expression>}*
```

where the symbol results is used if more than one value is to be returned.

Normally a procedure returns to the calling program after the last statement in its statement part has been executed. If it is desired to return from any other part of the procedure, then the return statement must be used.

Example: Suppose the coefficients of a polynomial are stored as the sequence *coeff*. The function *eval* below evaluates the polynomial for a particular value of the indeterminate *x*. *Length* is a built-in function which gives the number of terms in a sequence.

```
function eval(coeff, x);
   begin n=length(coeff);
         val=0;
         for i=0 to n-1 do
          val=val*x+coeff[n-i];
         end;
         result val;
   end,
```

§11. Input/Output

Cayley possesses high-level input/output facilities which permit the reading and writing of highly structured information such as subgroup lattices. Apart from read and write statements, which are chiefly designed for reading and writing mass storate files, we have a read function and a print statement which provides for the printing of any Cayley data type or attribute in an attractive format which we hope is easily comprehensible to the algebraist.

<read function designator> ::=
read(<file identifier>,<file format>,<input type>)

<input/output statement> ::=
print<print list>|
read(<file identifier>,<file format>)
<attribute list>|
write(<file identifier>,<file format>)
{<attribute list> / <variable list>}
{read / write}(<file identifier>,<file format>)
<algebraic type class>

<print list> ::= {<expression> / <string>}{,{<expression> / <string>}}*|
<attribute>{,<parameter list>}|
<algebraic type class>

<input type> ::=
<algebraic type class>|<element type class>|
boolean|integer|character

<element type class> ::=
group element|ring element|field element|vector|
module element

<attribute list> ::= <attribute>{,<attribute>}*

File identifier designates the file to be read or written, while file format specifies how the information is to be read or written on the file, e.g. in binary or coded form. The read function with algebraic type class in place of type is used to input a set of attributes delimited by begin and end. The attributes are typed in the form

<attribute>:<expression>

or

<attribute>:<attribute definition expression>

Information of a particular type is input in the usual expression form.

Examples:

i = read(5, coded, integer)

The input record may contain an integer (e.g. 9), a set of integers (e.g. [2,-1,7,16]) or an integer sequence (e.g. {2,3,5,7,11}). Depending upon which is the case, *i* will be assigned an integer, a set or a sequence as its value.

t = read(5, coded, group element)

The input record may contain either an element, a set of elements, a sequence of elements or a group generated by designated elements. In each case the element or elements must belong to the current group. Thus if the input record contains [1,a,b,a b] or [1,(1,2),(1,3),(1,2,3)] then t is assigned a set of elements.

group: K = read(5,coded, group)

The input record is expected to contain a set of attribute assignments delimited by begin and end, e.g.

```
begin;
   name: S4;
   degree:4;
   generators: a=(1,2),b=(2,3,4);
   relations: a↑2=b↑3=(a*b)↑4=1;
end;
```

The read and write statements allow any expression, attribute, set of attributes, or complete algebraic structure to be written to a file and retrieved when desired.

Examples:

```
write(7,binary) generators, elements, subgroups
read (7,binary) generators, elements, subgroups
write(7,binary) set1, centre, sgp
read (7,binary) mengel,u1, u2
write(7,binary) group
```

The last statement will cause the current group to be written.

The normal method of outputting information in human readable form is by using the print statement. This can be used to print the values of variables, attributes and strings. A form of print is provided which allows the user to selectively print parts

of certain attributes. Thus for example, instead of printing all the subgroups of a group, one could specify that only a representative from each conjugacy class be printed. If the print list is replaced by an algebraic class type, then all attributes for the current structure of that class type will be printed.

Note that the special strings $N, $P, $$ and $' are used to denote new line, new page and the characters $ and ' respectively.

Examples:

```
print  a*b, a↑-1,b↑a
print  'element = $N' a*b*a
print  subgroups
print  subgroups, reps
print  classes, order=2, reps: no, elts:yes
```

The second last print statement contains the parameter *reps* which indicates that only representat--ive subgroups from each class are to be printed.

§12 Programs

A program is a sequence of statements, procedure declarations and funtion declarations. Execution is terminated by the quit statement.

§13 The Cayley Library

The power of Cayley as a problem solving tool in group theory stems from its extensive library of built-in functions. Included among these are functions for computing elements, classes, subgroups, various types of series, Sylow subgroups, normalizers and centralizers, etc. Functions for combinatorial group theory include the Todd-Coxeter, Reidemeister-Schreier, and nilpotent quotient algorithms. Functions testing for a wide variety of group and subgroup properties are provided. The current Cayley library is described in Cannon and Neubüser [3].

§14 A Sample Program

In Figure 1 we give a function *Sylow* that computes the Sylow p-subgroup of a group. The algorithm we use first searches the group for an element of order p and, then starting with the subgroup generated by this element, successively constructs a chain of p-groups by cyclic extensions, which eventually terminates with the Sylow p-subgroup. Potential extending elements are sought in the centralizer of an element of order p in the centre of the current p-subgroup.

Built-in functions *order*, *centralizer*, *normalizer*, *centre* and *transversal* are used. A built-in function is actually provided for calculating the Sylow p-subgroup but the algorithm is given here coded in Cayley as a demonstration of some of the features of the language.

Acknowledgements

I would like to express my gratitude to P.T.J. Nicholson for contributing many ideas to the detailed design of Cayley and for producing a first implementation. Of the numerous other people who made contributions to the design of Cayley I would particularly like to mention V. Felsch, J. Neubüser, U. Schöenwaelder and N. Wirth. I would also like to express my appreciation to G. Havas and J.B. Hext for critically reading a draft of this manuscript. However, responsibility for the design and its description in this paper is mine alone.

The implementation of a teaching subset of Cayley was supported by a grant from the Australian Council for Research and Development in Education. The implementation of the remainder of the language and those built-in functions written in Sydney was supported by the Australian Research Grants Committee.

References

1. John J. Cannon: A general purpose group theory program. Procs Second Internat. Conf. Theory of Groups, Canberra, 1973, 204-217.

2. John J. Cannon: A draft description of the group theory language Cayley. Computer-Aided Mathematics Project, Technical Report No. 12, Department of Pure Mathematics, University of Sydney, Sydney, April, 1976.

3. John J. Cannon and Joachim Neubüser: The GROUP system library.

4. Volkmar Felsch: A machine independent implementation of a collection algorithm for the multiplication of group elements. These proceedings.

5. N. Wirth: The programming language Pascal. Acta Informatica 1, (1971), 35-63.

FIGURE 1.

A Sylow subgroup algorithm coded in Cayley

```
function Sylow (G,p)
"construct the Sylow p-subgroup of group G"
begin    group: G;
         k = expon(order(G),p);
         if k = ϕ then result <1> end;
        "look for an element of order p"
         for each element x do
                  j = order(x);
                  r = expon(j,p);
                  if r > 0
                           then subg = <x↑(j/p↑r)>;
                                    break
                           end

         end;
         "we now have a cyclic subgroup of order p↑r"
         done = [1];
         while r < k do
                  "find the centralizer of an element of order p in the centre"
                  c = centre (subg);
                  for each x in c-done do
                                    if order (x) = p
                                    then done = done ∪ <x>;
                                             c = centralizer (G,x);
                                             break
                                    else done = done ∪ [x]
                                    end;
                           end;
                  l = order (c);
                  if l = p↑j then result c end;
                  l = expon (l,p)
                  "p↑l is the order of the Sylow p-subgroup of c"
                  while r < l do
                           for each x in transversal (normalizer (c, subg), subg)
                                    do if p↑l mod order(x) = 0
                                             then subg = <subg, x>;
                                             break
                                    end;
                           end;
                           r = expon (order (subg), p);
                  end;
                  "we now have found the Sylow p-subgroup inside the centralizer c"
         end;
```

```
                result subg

end;

function expon (m,n)
"finds the largest integer r such that n^r|m"

begin       r = 0;
            while m mod n = 0
            do m = m/n;
               r = r+1;
            end;
            result r;
end;
```

Recursive Programming through Table Look-up *

Daniel P. Friedman
David S. Wise
Mitchell Wand

Computer Science Department
Indiana University
Bloomington, Indiana 47401

Abstract - The maintenance of arbitrarily large tables of previously computed values for functions on integer domains becomes practical when those tables are built using constructor functions which suspend evaluation of their arguments. Two styles of programming with such tables are presented. The first results from replacing recursive invocations within standard recursive function definitions with a reference into a table which is predefined to be all the possible results of the function. The second, more sophisticated, style requires that the table be defined strictly through a generation scheme. In either case the table may be available to the user as a data structure exclusive of the function definition with entries still being manifested only when they are actually used.

Keywords and Phrases - suspended evaluation, generation, streams, infinite lists, dynamic owns, LISP.

CR Categories - 4.13, 4.22, 5.7, 4.12

Introduction

The goal of this paper is to demonstrate that standard recursive definitions may easily be transformed to permit the specification of a table of arbitrary size. The table, built with "suicidal suspensions" [Friedman and Wise 1976], only manifests itself for values which have been explicitly used; these values remain easily accessible for later reuse. The values which are computed from an algorithm expressed in "top-down" recursive code are computed through a bottom-up computation sequence which is time-optimal for that algorithm in the sense of Berry [1976]. Such tables may also be specified by generating functions based on an unconventional formulation of traditional recurrence relations requiring the concept of a suspended structure.

The technique of purely recursive expression was popularized by McCarthy [1960] in the fundamentals of the language LISP, referred to here as pure LISP and elsewhere as LISP 1.0. As this language became more popular, different researchers with varying needs added more and more "enhancements" to the language; notable versions, like LISP 1.5, LISP 1.6, MACLISP, and INTERLISP found their way into common use. In this paper we build on recent results which provide the least-fixed-point semantics for pure LISP, demonstrating new programming styles available within pure LISP with the new semantics. These new styles provide a means for accomplishing many of the goals which motivated the expanded versions of LISP, without the addition of extra structures to the basic language.

The new programming style allows function application as the only control structure and lambda-binding of parameters as the only dynamic association between identifier and value. We provide static bindings, which are not explicitly part of pure LISP, in order to simplify the identification of globally defined functions and globally defined constants. These may be subsumed into classic definitions of pure LISP by regarding such bindings as part of the global association list which exists before run-time. These static globals will be declared explicitly with the use of the symbol "≡".

The concept of suspended evaluation is introduced for LISP by Friedman and Wise [1976] and is similar to lazy evaluation [Henderson and Morris 1976] in its theoretical impact. It is easily implemented by changing the semantics of the elementary functions car, cdr, and cons. Instead of placing the final values in the node allocated by cons, each unevaluated argument with the current environment is placed there in a distinguishable structure called a suspension. The value of a call on cons is unchanged. When either of the two projection functions is invoked on that node, the argument is evaluated in the preserved environment. The resulting value takes the place of the suspension in the node and is returned. As a result the LISP evaluator is invoked no more under this interpretation of elementary functions than under McCarthy's classic interpretation, and in

*Research reported herein was supported (in part) by the National Science Foundation under grant no. DCR75-06678 and no. MCS75-08145.

many instances the evaluator is called significantly less. Thus when the interpreter itself is interpreted under these semantics the least-fixed-point semantics for LISP results.

In this paper we study the effects of this interpretation on programming style and treatment of data structures. This treatment does not depend on a different interpreter, but only on the treatment of the structure manipulation functions car, cdr, and cons. The results generalize trivially to records [Hoare 1975] of more than two fields.

The LISP syntax used here is standard with two minor exceptions. Conditional expressions are presented as alternating sequences of predicates and selections with the commenting keywords if, then, elseif, and else interspersed in order to suggest the pairing which in traditional code is expressed by extra parentheses, creating the appearance of bogus function calls. We have shown elsewhere that under the new semantics cond can be just another function, as it appears in this syntax [Friedman and Wise 1976]. The other syntactic change is related to subscripting of arrays. The form <i lis> where i is an integer and lis is a list returns the ith element in lis.

Infinite Lists

Let us begin with a simple LISP function which no programmer would dare invoke under the old interpretation scheme. Successors is globally defined here as a function and naturals is defined as a constant.

```
(successors n) ≡ .
    (cons (addl n)
          (successors (addl n)) ) ;

 naturals ≡ (successors 0)  .
```

Assuming that the call to successors is not dispatched from the top level of LISP (which includes an implicit print function which traverses the result using car and cdr), we encounter no difficulty in the evaluation of naturals. Its value is a reference to a single node whose A-field, cAr field, is a suspension which would evaluate to 1 if ever coerced and whose D-field, cDr field, is a suspension which would evaluate to another node initially filled with similar suspensions, were it coerced. This structure of naturals is, of course, never seen by the user. Naturals is perceived as a structure to be manipulated, and all reasonable manipulations are well-defined.

For instance, one might ask to see the value of <4 naturals>, the fourth element on the list, sometimes phrased (cadddr naturals). As the interpreter finishes three successive cdr's and then a car, four coercions occur, creating three new nodes and filling in four fields permanently. The result is that 4 is returned, and a change to the invisible physical data structure takes place. A later invocation of <4 naturals> creates no new nodes and causes no coercions. We say the actual data structure is invisible because it reacts to manipulation as if car and cdr were probes into a delicately balanced physical system: whenever a field is accessed its final contents appear.

```
(terms n) ≡ (cons (reciprocal (square n))
                  (terms (addl n))) .
```

This sequence should be familiar to analysts: the sum of the elements (terms 1) converges to $\pi^2/6$. If one made the mistake of printing (terms 1) then the output would begin

(1.0 0.25 0.1111 0.0625 0.04 etc.

until the capacity of the local machine were exhausted. The sequence is infinite so of course this behavior is to be expected. Under the old interpretation of cons nothing would be printed at all because the traversal of a result by print does not begin until that result is completely manifested as a data structure. Because the suspending cons creates the data structure as the top-down traversal proceeds, we have the net effect of top-down evaluation of recursive code.

Landin [1965] defines a stream as, in effect, a cons which evaluates its first argument but suspends its second. If terms were defined using the stream model then the result of printing (terms 1) would be the same finite prefix of an infinite list as indicated above. The difference is apparent when one considers <3 (terms 0)>. Under the stream model the entire prefix of the list (terms 0) must be coerced up to and including the third element; this coercion diverges because the first element diverges. Under the completely suspending cons, <3 (terms 0)> converges to 1/4 or 0.25 because the value of $1/0^2$ is not coerced in obtaining the value which is the third on the list.

Global Tables

With the elementary concept of infinite lists it is possible to define global structures which are conceptually unbounded but which are, in fact, of a finite length determined by past use. The effect is similar to the dynamic own array of ALGOL 60, but without the associated difficulties. Consider the following function for the ith Fibonacci number, which assumes unused own variables are always initialized to zero.

```
integer procedure Fibonacci(n);
  value n;  integer n;
  begin
    own integer array fib[1:n];
    Fibonacci := (if n < 0 then 0 else
                  if n = 0 then 1 else
                  if fib[n] ≠ 0 then fib[n]
                  else (fib[n] :=
                        Fibonacci(n-2) +
                        Fibonacci(n-1) ))
  end.
```

This procedure has the property that no value of Fibonacci is computed more than once. Thus the sequence of states of the array fib is a time-optimal bottom-up computation sequence in the sense of Berry [1976]. This property is characteristic of the programs we consider.

The procedure runs in time 0(n) if the operation of array access is presumed to take constant time. As elegant as that behavior appears when compared with $0(2^n)$ behavior of the classic recursion

```
else (Fibonacci(n-2) + Fibonacci(n-1))
```

it is not a fair bound when the problems of dynamic own arrays are considered. If such an array grows larger and larger with successive calls to its enclosing block, the system will either be forced to recopy a sequentially stored vector into a larger buffer, or it will be forced to successively link additional storage onto the pre-existing structure on successive calls. In either case access into such a structure of size n should be charged time 0(n), and the net run-time of Fibonacci is $0(n^2)$.

Compare the ALGOL-like code above, with its gross assumptions on initialization of variables and its exclusively local array, to the LISP globals developed below. Due to subscripting conventions,

```
<n fibtable>
```

is equal to

```
(fibonacci (subl n)) .
```

```
fibtable ≡ (fibtail 0);
(fibtail n) ≡ (cons (fibonacci n)
                    (fibtail (addl n)) ) ;
(fibonacci n) ≡ (cond
    if (lessp n 0) then 0
    elseif (lessp n 1) then 1
    else (sum (fibonacci (sub2 n))
              <n fibtable> )   ) .
```

In order to replace the recursive call (fibonacci (sub2 n)) by a table reference we must be sure that the table entries always exist. This can be accomplished by a minor alteration in the bases:

```
(fibonacci n) ≡ (cond
    if (lessp n 0) then 0
    elseif (lessp n 2) then 1
    else (sum <(subl n) fibtable>
              <n fibtable> ) ) .
```

The structure fibtable is accessible throughout the system. Atoms may be lambda-bound to it; a wasteful user might even attempt to see it, abusing LISP for a peek at its prefix. The time to compute <n fibtable> is bounded by $0(n^2)$, taking into account the 0(i) time needed to traverse to the ith element on a list.

We believe that the implementation of infinite structures through suspending cons is far sounder than through other programming styles. The technique of dynamic own arrays is far more powerful and correspondingly more complex than the LISP approach. The contents of dynamic own arrays may be changed at any time while the values within a LISP structure are constant with respect to the binding to the structure, hence pre-ordained, if not already extant. Some care is necessary in constructing the array so that recurrence relations which reference other entries in the array do in fact converge, but this is the same as determining that a highly recursive function definition defines a total function. Such references parallel recursive function calls, asking only a natural extension of LISPers' already refined concepts of recursion and cons, whereas the manipulation of dynamic own arrays require some appreciation of implementation details, such as initialization conventions.

Multi-dimensioned Tables

A recursive (numeric) function of several parameters may use a similarly structured table to retain its values. The dimension is determined by the number of arguments. Interestingly, depending upon the recursion pattern, some entries in this table may never be created no matter what arguments are supplied to the associated function. A familiar example is available in the combination (binomial coefficient) function. Defined in ALGOL and taking into account a well known symmetry it appears as

```
integer procedure Choose (m, n);
  value m, n; integer m, n;
  Choose := (if n < 0 then 0 else
             if n = 0 then 1 else
             if n+n > m then Choose(m, m-n)
             else Choose(m-1, n) +
                  Choose(m-1, n-1)   .
```

This recursion is that often associated with Pascal's triangle, which is the table of values associated with Choose. The subscripting convention again requires that <n <m triangle>> is equal to (choose (subl m) (subl n)); a matrix is represented as a list of vectors so that this expression refers to the nth entry in the mth vector.

```
triangle ≡ (rows 0) ;
(rows m) ≡ (cons (tail m 0)
                 (rows (addl m)) ) ;
(tail m n) ≡ (cons (choose m n)
                   (tail m (addl n)) ) ;
(choose m n) ≡ (cond
    if (lessp n 0) then 0
    elseif (zerop n) then 1
    elseif (greaterp (sum n n) m)
           then (choose m (diff m n))
    else (sum <(addl n)<m triangle>>
              <n <m triangle>> ) ) .
```

As before triangle is a globally defined list which may be inspected or traversed by any function which accesses it as a free variable, but the bindings associated with it cannot be permanently altered. It may be treated as an infinite two-dimensional array, but only part of it will ever be actually created by any single program. In fact, if its only use is as a free variable to choose then only the quarter below a "south-by-southeast" diagonal will ever have manifested values (where the "southeast" diagonal is the main diagonal and all entries above it are zero upon access. See Figure 1.)

To see more clearly which values are created, let us consider some examples. One can evaluate (choose m 0) and (choose m m) without causing any more of triangle to appear than the single node (containing suspensions) which was originally allocated to be its value. Assuming that no entries of triangle have yet been coerced and that $0 < n < m/2$ then a call of (choose m n) will cause the

$$n(m-n) + 2(m-1)$$

nodes to be allocated which build an upper-left corner of triangle (Figure 1) with a capacity for

$$n(m-n) + (m-1)$$

values, but only

$$n(n+1) - \frac{n}{2}(3n-1) - 1$$

values are actually entered. Surprisingly the entry in <(addl n)<(addl m) triangle>> is not made as a result of this call even though the value is at hand.

A dramatically different behavior occurs if the programmer accesses <n <m triangle>> directly rather than calling (choose (subl m) (subl n)). In this case the value is entered into the table and then returned. In this way the zero entries above the main diagnonal of triangle may be called into existence as can never happen when the structure is only accessed via the function choose.

Table accesses are a convenient substitute for recursion whenever the function is commonly used and has the integers as its domain. It is particularly convenient when the recursion pattern is irregular so that the necessary recurrent values are not easily predicted and are supplied through additional clever arguments. Such a recursion pattern is analogous to course-of-values induction.

The programmer, having established a table of the values for a function, is free to access the table or to call the function. If the function contains clever code, such as the symmetry handler within choose, then our advice is to stick with the function. However, if the function requires multiple accesses into the table then the expense of accessing the dynamic data structure can drive the running time bound up to polynomials of degree higher than that warranted by the recursion pattern. In the next section we provide a way to avoid this cost.

Generative Programming

If the programmer uses a table to preserve the values of the function and has that table around at the same time as a potentially useful data structure, then he can define the values in terms of the table alone. We claim that the expression of functions as generated tables is naturally related to the kind of programming exhibited above and we shall demonstrate some advantages of that form of expression.

We briefly present generative forms for the functional tables discussed above.

```
 fibtable ≡ (fibtail 0 1) ;

(fibtail m n) ≡
       (cons n (fibtail n (sum m n)))
```

Now the function call (fibonacci n) is replaced by a table reference <(addl n) fibtable> . Since only one table traversal is performed, the time for all traversals collapses to 0(n) and so the total time becomes linear.

```
 triangle ≡ (pascal (cons 1 (zeros))) ;

(zeros) ≡ (cons 0 (zeros)) ;

(pascal row) ≡
    (cons row
          (pascal (cons 1
                        (pairsum row) )) );

(pairsum row) ≡
    (cons (sum (car row)
               (car (cdr row)) )
          (pairsum  (cdr row)) ) .
```

As before <n <m triangle>> is synonomous with (choose (subl m) (subl n)) but without the other definition of choose this computation method, including a single subscripted access, is the only way to recover the binomial coefficients.

Expression of functions through generated tables also allows the table to be built locally and maintained through a temporary lambda-binding instead of through a global declaration. In the examples of the earlier section we accessed the table from within the function which was used to determine other entries in it. This very nearly circular definition was possible because the referenced object was declared to be global at the expense of its being static in interpretation (although not in physical structure). When the same table is expressed generatively its generator may be passed as an argument for local binding and interpretation only within a temporary environment. Upon exit from that inner environment the structure is recovered. A function call like <7 (fibtail 0 2)> returns

26, having established a table for a different Fibonacci sequence which is partially manifested and then lost.

A property of the table generators above is that they do not use the function cond in their definitions. That is a symptom of a very regular recurrence which is not always possible.

As an example of table generated using cond within the code we offer the following closed form for the Sieve of Eratosthenes which generates a list of all the prime numbers without a multiplicative operation. (Henderson and Morris [1976] offer a similar example.) The critical function is removemult which removes values of mult + i·inc for all i from the monotonic list of integers lis, where inc, mult, and lis are its parameters.

```
 primes ≡ (sieve (successors 1)) ;

(sieve nums) ≡
  (cons (car nums)
        (sieve (removemult
                        (car nums)
                        (double (car nums))
                        (cdr nums) )) ) ;

(removemult inc mult lis) ≡ (cond
    if (lessp (car lis) mult)
     then (cons (car lis)
                (removemult inc
                            mult
                            (cdr lis) ))
    elseif (greaterp (car lis) mult)
     then (removemult inc
                      (sum mult inc)
                      lis)
    else (removemult inc
                     mult
                     (cdr lis) ) ) .
```

The function call (double n) meaning (sum n n) might be replaced by (square n) by a programmer tolerant of multiplication and familiar with number theory. So the primitive recursive function (prime i) which returns the ith prime is merely <i primes> ; the recursive definition of this function is rather complex without a table or with a table reference used only as a replacement for a recursive call as in the previous section.

Conclusions

The use of an arbitraily large table made possible by the suspending cons was shown to be convenient in efficiently expressing recursive numeric procedures. It is often very easy to replace recursive function calls by accesses into a global structure which preserves the results of earlier calls usable for just the cost of finding them. At this level, such tables may be simulated by dynamic own arrays in block structured languages but at some expense in expression and compiler difficulties.

A more sophisticated use of these structures, although one which asks more than classic recursive definitions of functions, is their use as the explicit function definition. In order to meet this use, the arbitrarily large structure must be expressed as a result of a generating function which is in closed form (exclusive of the table). The computation performance will be slightly more efficient in this form but the required expression may be a problem because such definitions are not in common use. The user can achieve improvements in performance with either choice if the functions are commonly used or highly recursive, just by avoiding wasted effort.

Acknowledgement

We thank Cynthia Brown for her careful reading of the manuscript and for her suggestions which improved it.

References

G. Berry. Bottom-up computation of recursive programs. Revue Francaise d'Informatique et de Recherche Operationaelle 10, 3 (March, 1976), 47-82.

W.H. Burge. Recursive Programming Techniques, Addison-Wesley, Reading, MA (1975).

D.P. Friedman and D.S. Wise. Cons should not evaluate its arguments. Third International Colloquium on Automata, Languages and Programming, Edinburgh University Press (July, 1976).

P. Henderson and J.H. Morris, Jr. A lazy evaluator. Third ACM Symposium on Principles of Programming Languages (January, 1976), 95-103.

C.A.R. Hoare. Recursive data structures. International Journal of Computer and Information Sciences 2, (June, 1975), 105-132.

J. McCarthy. Recursive functions of symbolic expressions and their computation by machine - I. Comm. ACM 3, 4 (April, 1960), 184-195.

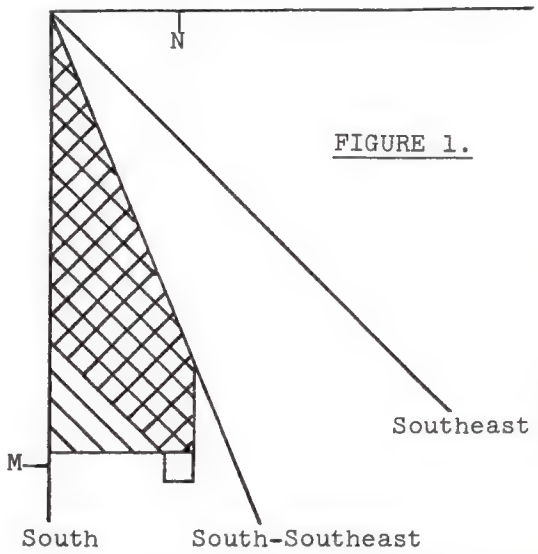


FIGURE 1.

Allocations coerced by (CHOOSE M N) . The hatched area indicates nodes allocated due to coercion of cdr fields. The cross-hatched-area indicates nodes whose car fields are evaluated as well.

Symbolic Evaluation of Programs: A Look at Loop Analysis*

T. E. Cheatham and J. A. Townley
Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts 02138

I. INTRODUCTION

The past several years have witnessed a growing concern with the high cost of software development and maintenance. The usual reaction to this has been administrative -- to organize programming activities better (the Chief Programmer Team); to improve the structure of programming languages, and thus, hopefully, the programs written in them; or to propose that formal "specifications" be developed for a program before the coding of it in some programming language commences.

We certainly agree that the infusion of "structure" into the programming process is an important step to take. However, we also believe that the ultimate solution to the problem lies in taking an entirely new approach to the process of programming. To this end, we are engaged in a relatively long term project to develop what we have termed the Harvard Program Manipulation System (PMS). The PMS will provide a variety of tools and facilities for program development and maintenance, with the program and its documentation being in machinable form from the earliest stages of sketching the functions and inter-relationships of modules through the stages of refinement and selection of representations for data and operations which ultimately result in the actual operational program. The maintenance of the program would also be carried out with the aid of the PMS in an environment in which the complete history of the program's development would be available to aid in making the changes required in the maintenance process.

The PMS will include the usual kinds of programming aids such as editors, a file system for programs and documentation, a query facility, debugging aids, and so on. However we also propose to provide a number of tools which are not so usual. We briefly describe some of these tools in the following paragraphs (but leave to future publications more detailed explanations of their design).

1. Source-to-Source Transformation Facility

There is a limit to the kinds of program improvement (optimization) which can be accomplished by ordinary compiling methods. A source-to-source transformation facility can provide for "higher level" (and high pay-off) transformations such as loop cleavage, loop fusion, replacement of recursive constructions by iterative equivalents, constant propagation, elimination of the need for temporary arrays, and so on.

2. Program Analysis and Fault-Finding

The usual debugging aids essentially monitor the execution of a program, thus enhancing the user's ability to control and observe what transpires during the evaluation. We propose to augment this with a program analyzer which both answers questions about the program and verifies that certain operations are valid.

The kinds of questions which can be posed to the program analyzer include questions concerning the conditions under which control can reach certain points, the constraints which must be true at various points in the program, the different values which might be associated with various variables, and so on. Examples include such questions as "what constraints are true of some variable, V, used free in procedure P?", "can control ever reach this point?", "is the selection X[I] valid at some particular program point?". It is hoped that the program analyzer can both provide more confidence in the correctness of some program (particularly with respect to the details of loop termination, selections being valid, modes of actual parameters matching expected modes, and so on) as well as aiding in the generation of test data (for example producing test data that will insure that certain control paths are taken).

*Research supported by the Advanced Research Projects Agency under contracts N00039-76-C-0168 and F-30602-74-C-0032.

3. Selection of Data Representations

In many applications, one of the difficult problems is the selection of appropriate representations for high level constructs, such as sets or sparse arrays. The difficulties arise both from the fact that the information which is required to make a good (or optimal) choice is difficult to obtain and from the fact that a given programmer may only know about a limited repertoire of possibilities. We intend to provide a facility which will help in this choice by gathering data relevant to the decision from inspection of a program and by providing a rich inventory of possibilities with the means for guiding the choice from an inventory based on the data gathered.

4. Cost Analysis

The usual means to analyze (dynamically) the cost of a program in terms of time is to insert probes into the running program. Space analysis is most typically done by compiling and measuring the resulting size of competing implementations. In the PMS we hope to be able also to develop cost expressions for programs or parts of programs based on static analysis of the programs. These cost expressions will take the form of probability measures involving parameters such as sizes of arrays, the probability of some condition being maintained, and so on.

Clearly each of the tools cited above requires a reasonably sophisticated "understanding" of the program and its effects. The Symbolic Evaluator (SE) which is the subject of this paper, provides this "understanding". The purpose of the SE is to take a program and to develop a program data base. This program data base includes a "shadow" of the original program plus an inventory of facts about the program. The "shadow" essentially provides an "extension" of each construct in the program where the modes, values and other pertinent facts are recorded. The values are, of course, symbolic values which are functions of the actual parameters which would be input and of various conditions which arise during the course of evaluation. The inventory of facts about a program might well include such items as "the quantity N must satisfy $N \geq 1$ if a selection fault is to be avoided in loop L," or "the number of times loop R is repeated is

$$\min\{j \mid P(j) \text{ OR } j=N_1\}$$

The programming language which we are employing to develop the system and for which the system will provide the analytic and manipulative tools is the EL1 programming language [2,3]. EL1 is a sophisticated programming language providing for real, integer, boolean, and character data, plus sequences, structures, and pointers. The language is "hard typed" in the sense that each variable has a mode and size which is fixed at the time it is declared or allocated. Externally EL1 looks similar to the ALGOLs with block structure, declarations, iteration statements, and so on. The storage management includes both a stack for local data and a heap for pointer objects.

A complete description of the Symbolic Evaluator is beyond the scope of a single paper. What we shall do here is describe the techniques being employed for loop analysis. Following this we provide an overview of the structures employed to represent the result of a symbolic evaluation of some program and describe the basic modules of the symbolic evaluator. We end with a brief status report.

II. LOOP ANALYSIS

The heart of the symbolic evaluator is the loop analysis component. Given some loop, we develop equations for the values of all quantities which change in the body of the loop as functions of a formal variable, k, whose value is the current cycle number, plus other quantities which are invariant in the loop. We also develop expressions for the number of cycles taken (actual number if possible or maximum number if the actual number is conditional).

With most loops we cannot, of course, actually solve for the values of all variables as a simple function of the inputs. However, we can usually develop expressions which permit subsequent determination of the validity of certain constructs, by, for example, a program analyzer or source-source transformation facility.

Rather than provide a detailed description of the loop analyzer, we shall sketch the kind of processing involved by looking at three examples and discuss what the loop analyzer will deduce about their behavior. First, however, some comments about notation. We will describe each loop in EL1 with some comments which hopefully render it understandable to those who do not know EL1. We then describe the results of the loop analysis using equations. These equations are internally represented as EL1 expressions with certain variables interpreted as symbolic values. Throughout we use the notation I_k to denote the value of the quantity (variable) I at the beginning of the kth cycle of the loop (where k is a formal variable representing the cycle index). Each quantity which might change during the evaluation of some cycle of a loop is assigned such a value. At the end of a cycle, we will have the values of all variables represented as functions of the initial value of variables in that cycle plus other quantities which are invariant in the body of the loop. The values are represented as recurrence relations or recursion equations which we try to solve to obtain closed form expressions for the values of the quantities after a general cycle of the loop. Other notations will be commented upon as they are introduced.

Example 1.

Consider the following very simple program fragment:

```
DECL S:INT;
DECL I:INT BYVAL 1;
REPEAT
     S <- S + X[I];
     I <- I + 1;
     I GT N => S;
END;
```

Here S and I are declared to be integers with S initially zero (by default) and I initially one. In the body of the loop (REPEAT ... END) S is incremented by the Ith element of the vector X (presumed to be a sequence of integers), I is incremented by one, and then I is compared to N; if I exceeds N then the loop is terminated with result S; otherwise we cycle again.

Let X_1 and N_1 denote the values of X and N just prior to the loop body (S and I clearly having the values 0 and 1), and let I_k and S_k denote the values of I and S (the only quantities which change in the loop) at the beginning of the kth cycle of the loop. The symbolic evaluation of the loop body would then result in the following values (symbolic expressions) associated with S and I.

Quantity	Initial	Value Beginning of Cycle	End of Cycle
S	0	S_k	$S_k+X_1[I_k]$
I	1	I_k	I_k+1

Further, the "cycle again" condition would be

$$CC: I_k + 1 \text{ LE } N_1$$

Given this data, the loop analyzer then tries to obtain equations which represent the values of various quantities at the beginning of the kth cycle. Assuming that we are to cycle again, the symbolic expressions for I provide the recurrence relation

$$I_1 = 1$$
$$I_{k+1} = I_k + 1$$

which can be solved directly, yielding

$$I_k = k$$

(i.e., the value of the variable I at the beginning of the kth cycle of the loop is k.) Having solved for I_k we substitute this solution for the occurrence of I_k in the expression of S and obtain the recurrence relation

$$S_1 = 0$$
$$S_{k+1} = S_k + X_1[k]$$

Since X_1 is invariant in the loop we can solve this to obtain

$$S_k = \text{finite\sum}(j, k-1, X_1[j])$$

where finite\sum(p,N,E) is a function taking a special "dummy" symbolic value p as first argument, an integer valued quantity as second, and some expression (usually involving the dummy variable p) as third; it denotes the sum from 1 to N of the values E with 1, 2, ... , N successively substituted for p in E. Finally, we would like to determine how many cycles the loop takes. The exit condition (substituting k for I_k) is

$$k + 1 \text{ GT } N_1$$

Now surely at least one cycle is started; if $N_1 \leq 1$ then exit occurs on the first cycle. If $N_1 \geq 1$ then clearly N_1 cycles are started (i.e. exit occurs on the N_1th cycle). Denoting by k_L the value of k on the last cycle started we thus have

$$k_L = \max(1,N_1)$$

We also note that the values of I and S following the loop are (by substitution of k_L for k in the above equations)

$$I = k_L + 1$$
$$S = \text{finite\sum}(j, k_L-1, X_1[j])+X_1[k_L]$$

which simplifies to

$$S = \text{finite\sum}(j, k_L, X_1[j])$$

A final remark on this loop: assuming that X had length N_1 (with valid subscripts 1, 2, ... , N_1), the user employing the program analyzer could be informed of the possible selection fault in this loop which would occur if $N_1 \leq 0$; he might choose to remedy this by modifying the program to check the value of I prior to the selection or by providing the assertion (or permitting the deduction) that $N_1 \geq 1$, with the result that k_L would then simplify to

$$k_L = N_1$$

Example 2.

Consider the following loop:

```
DECL I:INT BYVAL 1;
REPEAT
     I GT LENGTH(Y) OR Y[I] = 0 => X;
     X[I] <- Y[I];
     I <- I + 1;
END;
```

Here we are copying the vector Y into the vector X (assumed to have the same length as Y), stopping either when we have copied all the elements or when we find a zero element in Y. The symbolic evaluation of this loop would commence with the following values associated with X and I (assuming $X = X_1$ initially)

Quantity	Initial	Beginning of Cycle
X	X_1	X_k
I	1	I_k

If exit did not occur the value of I at the end of the cycle would be

$$I_k + 1$$

and the value of X would be

$$rec\backslash const(j,N_1,cond(j=I_k,Y_1[I_k],X_k[j]))$$

where Y_1 is assumed to be the initial value of Y and N_1 denotes the length of Y (a symbolic value constructed when Y was created). Two new functions are introduced by this example - rec\const and cond. cond(P,A,B) is a function which returns A if P evaluates to true and B otherwise. rec\const is a function which takes three arguments: the first is a dummy variable, the second an integer, and the third an expression usually containing occurrences of the dummy variable.* The interpretation, for example, of

$$rec\backslash const(j,10,j)$$

is that vector of length 10 whose jth component is the integer j (the APL iota function). The rec\const( ) value for X above indicates that the value of the jth component of X is $Y_1[I_k]$ if $j = I_k$ and what it was at the beginning of the cycle otherwise.

The exit condition is (taking N_1 as the value of the length of Y)

$$I_k \text{ GT } N_1 \text{ OR } Y_1[I_k] = 0$$

As with the first example, we can directly solve for I to obtain

$$I_k = k$$

and then substitute this for occurrences of I_k in the other expressions.

The number of cycles taken in this loop is somewhat more complicated. We must use a minimization operator to capture the conditionality of the number of cycles, obtaining

$$k_L = \min(j,Y_1[j] = 0 \text{ OR } j = N_1 + 1)$$

where j is a dummy variable. The value of X is obtained by analyzing the rec\const(...) value for X at the end of the cycle. We observe that only the kth component of X is modified on the kth cycle (in which we do not exit) and thus deduce that the value of X at the time we exit the loop is

$$X = rec\backslash const(j,N_1,cond(j<k_L,Y_1[j],X_1[j]))$$

If at some later time we had the further information** that the vector Y contained no zero elements, we could simplify the above equations to obtain

$$k_L = N_1 + 1$$

and

$$X = rec\backslash const(j,N_1,Y_1[j])$$

which would further simplify to

$$X = Y_1$$

*Actually rec\const takes vectors as its first two arguments, but we will not consider its more general form here.

**Gained, for example, by user supplied assertions or by deduction from previous manipulations involving Y.

Example 3.

Consider the following loop for copying selected elements from some list, where we assume that a LIST is a pointer to a LIST\EL. Each LIST\EL has two components: ELEM, the element, and NEXT a pointer to the next LIST\EL.

```
DECL R,RL:LIST;
DECL L:LIST BYVAL T;
REPEAT
    L = NIL => R;
    P(L.ELEM) ->
        BEGIN
            R = NIL =>
                R <- RL <-
                    ALLOC(LIST\EL OF
                        L.ELEM,NIL);
            RL <- RL.NEXT <-
                ALLOC(LIST\EL OF
                    L.ELEM,NIL)
        END;
    L <- L.NEXT;
END;
```

Here* we declare R and RL to be new LISTs, initially NIL (the value of the null or empty pointer) and L to be a copy of the LIST, T, to be copied. R will be the resulting copy and RL will point to its current last element to enable new elements to be added at the end. The predicate P is applied to the ELEM field of the current element of the list to be copied; if TRUE a list element is created (allocated with its ELEM field assigned the value of L.ELEM and NEXT field NIL) and either it initializes R and RL if it is the first to be added or is added to the end of the copy.

The value for L at the end of a cycle, assuming we cycle again, is

$$L = L_k.NEXT$$

The values for R and RL at the end of a cycle are rather more complex, being dependent upon two predicates. Assuming we cycle again, this dependence is as follows:

Predicates		Values of R and RL	
$P(L_k.ELEM)$	R_k=NIL	R_{k+1}	RL_{k+1}
TRUE	TRUE	loc(1)	loc(1)
TRUE	FALSE	R_k	loc(2)
FALSE	TRUE	R_k	RL_k
FALSE	FALSE	R_k	RL_k

*The EL1 construct

BEGIN p => s_1; s_2 END

is interpreted as follows: if p evaluates to TRUE then s_1 is evaluated and is the result of the construct; otherwise s_1 is not evaluated, but s_2 is evaluated and is the result of the construct. The construct

p -> s

is interpreted as follows: if p evaluates to TRUE then s is evaluated; otherwise s is not evaluated.

Here loc(1) and loc(2) represent addresses of (i.e. pointers to) newly created locations whose contents are each LIST\ELs whose ELEM field is L_k.ELEM and whose NEXT field is NIL.

Finally, the exit condition is

$$L_k = \text{NIL}$$

The conditionality of values plus the sharing induced by employing pointers makes this loop rather more difficult to analyze than the two preceding examples. The first step in the analysis is to attempt to determine the basis for the iteration. The exit condition depends only upon the value of L; L takes as successive values

$$T_1,\ T_1.\text{NEXT},\ T_1.\text{NEXT.NEXT},\ \ldots$$

(where T_1 is the value of T at the beginning of the loop). We solve for L_k by mapping the successive selections of the NEXT field onto the integers via the function

$$L_k = \text{list\element}(k, T_1, \text{NEXT})$$

which denotes the kth element of the list T_1 following NEXT pointers. The number of cycles taken is then determined employing the function list\length as

$$k_L = \text{list\length}(T_1, \text{NEXT}) + 1$$

where list\length(T_1,NEXT) is the number of non-NIL elements found by commencing with T_1 and following NEXT links.

Now let us consider the problem of determining what happens to R and RL in the loop. In general, there are two kinds of conditionality which we look for. The first is a condition which, if true, results in no change in the loop (except to the cycle index and variables dependent just upon it). The second is a condition which "flips" the behavior of the loop in the sense that the loop behaves in one (stable) fashion prior to the condition maintaining and then behaves in another (but stable) fashion once the condition maintains.

In this loop, the condition P(L.ELEM) is clearly of the first kind since if the condition is false nothing changes (save the cycle index and associated quantities). We therefore conclude that this condition is going to effectively identify some subset of the original list T_1 over which we are cycling. We can therefore effectively ignore P and turn our attention to considering what happens to R and RL on the cycles* in which "something happens."

Suppose k_1 numbers the first cycle on which P evaluates to true. At the beginning of this cycle we have

$$R = R_{k_1} = \text{NIL}$$
$$RL = RL_{k_1} = \text{NIL}$$

and at the end we have

$$R = R_{k_1+1} = \text{loc}(1)$$
$$RL = RL_{k_1+1} = \text{loc}(1)$$

(specifically, R and RL share the same list element). We now observe that the condition R = NIL has "flipped" since it is not true for the next cycle and, since R does not change (directly) so long as R is not NIL, it will never become true again. Thus, we have essentially "factored" our loop into two loops with different behaviors. Now consider the "second" loop. The initial condition to this loop is

$$R_k = RL_k = \text{loc}(1)$$

and the value stored at loc(1) is a LIST\EL whose ELEM field is a copy of the ELEM field of the k_1th element of T_1 and whose NEXT field is NIL. Now R is invariant in the current loop (i.e. the value of R is still the address loc(1) after a cycle) but RL varies. Since the value of RL is the address of some location, we must study what happens to this location (as well as to the location named RL) if we are to deduce what happens during a cycle. We thus postulate the address of an arbitrary location, loc(*), as the value stored at RL at the beginning of the cycle. The allocation in the cycle results in a new location, say loc(3), whose value is a LIST\EL with appropriate ELEM and NEXT fields. The assignment to RL.NEXT represents a change in the value stored at location loc(*), specifically that its NEXT component is now loc(3). The assignment to RL results in its value also being loc(3).

At the end of a cycle, presuming we cycle again, the value at loc(*) is a LIST\EL whose ELEM component is the same as it was when we started the cycle and whose NEXT component points to the new location loc(3). We therefore conclude that a cycle of the loop on which we do not exit adds a new list element as the NEXT component of the value at the location loc(*) which we had at the beginning of the cycle. That is, we are constructing a list with a new element for each cycle (on which P(...) is true), using the NEXT component to "hook" the elements together. It remains to determine who "owns" the newly constructed list. Now the value loc(*) is that associated with RL at the beginning of a general cycle; hence the list constructed is "owned" by the location initially associated with RL, that is loc(1). We therefore conclude (loc(1) being the value of R) that the list constructed is the value of R at the end of the loop, and that RL will point at its last element.

The actual representation of the values of R and RL require the use of a list constructing function analogous to the rec\const( ) function employed to construct vector (or, in general, array) values. The function

$$\text{rec\list}(j, N, E, f)$$

denotes a list of N elements whose kth element is E with k substituted for j; the field f is employed to "hook" the elements together. To

*One way of looking at this is to imagine that the cycles taken are numbered $k_1, k_2, \ldots, k_q$ where k_j is the jth cycle on which P evaluates to true.

completely capture R we also need to indicate that subset of elements of T_1 which occurs in R (i.e. those for which P applied to the ELEM part is true). The function select provides for a mapping from some sequence 1, 2, ... , N to a subsequence k_1, k_2, ... ; specifically,

select(k,N,P)

returns the sequence k_1, k_2, ... where k_p is the value of k such that P is true for the pth time.

Let us summarize the basic strategy employed in doing loop analysis. The first step is to determine all the quantities which may change during some cycle of the loop. We then assign these quantities special symbolic values (e.g. we assume that I has the value I_k) at the beginning of the loop. We then symbolically evaluate the loop with the result that for each location which changed we have a (symbolic) expression for its value at the end of the cycle. This expression is a function of the symbolic values for (possibly changing) quantities at the beginning of the cycle (e.g. I_k), plus loop invariants (e.g. 1, LENGTH(Y_1), and so on). Some of these expressions are sufficiently simple recurrence relations for the value of the quantity that we can directly solve them for the value of the quantity as a function of the cycle number (k) and other quantities invariant in the loop. We can also obtain an expression for k_L, the number of cycles taken, sometimes as a simple closed expression (e.g. $k_L = N_1$) and more generally by employing the minimization operator.

III. OVERALL STRUCTURE

The following are the major structures which the SE builds as a result of the symbolic evaluation of some program.

1. Context Graph

The context graph is a directed acyclic* graph whose nodes correspond to points in the program where control branches or joins. Paths through the context graph then correspond to possible flow paths in the program.

2. The Environment

For each distinct variable in the program and for each quantity allocated in the heap we have an "environment entry" which consists of a location and a set of values. The location corresponds to the actual memory location in which values of the quantity would reside in an actual execution of the program, and contains such things as the declared mode** of the quantity, its scope, its dimensions (generally a symbolic expression) if it's an array, an indication of other quantities with which it may share storage, and so on.

The set of values associated with each location represents the values of the quantity in different contexts. Thus a value has a context part (identifying some node in the context graph plus a "time" number which sorts the sequence of assignments in a given flow block) and a value part. The value part may take several forms:

(a) A manifest constant.
(b) A symbolic value which represents a computation.
(c) A symbolic value which represents a "join" -- that is, the set of possible values the quantity may assume in some particular context.
(d) A symbolic value which represents the value of the quantity at the beginning of some loop (i.e. the I_k and the like discussed earlier).

3. The Shadow

The internal representation of EL1 programs (that is, the representation which the interpreter executes -- there being also compiled code which is not of interest here) is a list structure very similar to that employed in most LISP systems. The symbolic evaluation of an EL1 program results in what we call the shadow of the original. This is represented by a list structure isomorphic* to the original but which is augmented in several ways:

(a) Corresponding to an identifier in the original program we have a body of information which may include the location for that quantity, the (current) value of the quantity, and the current mode (for example the mode at a certain point in the program may be a specific one of the possibilities for a quantity with a generic mode).
(b) Corresponding to each computation in the original program we have the shadow computation augmented with the mode (and possibly the value) of the result.
(c) For blocks and loops (which in EL1 can have an arbitrary number of possible results -- one for each possible exit point) we associate a result list which identifies the possible results and provides the context for interpretation of each possible result.

The Symbolic Evaluator can, grossly, be divided into three components: the basic evaluator, the analyzer, and the simplifier. We discuss these in turn:

1. Basic Evaluator

The basic evaluator takes the internal representation of some EL1 program and essentially "walks" the list structure to construct the shadow of the original. At the same time it processes declarations to induce environment entries for the

*The graph is acyclic because of the way we represent loops (essentially as a single path of flow for the "general cycle") and because goto statements are not permitted in EL1.

**This may be generic in EL1.

*This is not quite true since certain "ghost computations" to explicate implicit conversions, dereferencing, and so on are added to the original.

quantities declared and processes assignments to induce the addition of the new value for the quantity assigned. The context graph is also constructed by the basic evaluator. When it encounters a loop, it sets up the special symbolic values for possibly changing quantities (e.g. the I_k), performs an evaluation of a (general) cycle, and then calls in the analyzer to "solve" the loop.

2. Analyzer

The analyzer has the job of determining the result of a loop and the values upon exit of variables changed within the loop. The data it is supplied consists of symbolic expressions describing the values of all quantities changed. The symbolic expressions involve the (symbolic) values of quantities at the beginning of the loop plus other quantities which are loop invariant. In addition, the exit condition for the loop is supplied as a symbolic boolean expression, again in terms of the beginning symbolic values of possibly changing quantities plus loop invariants. It then tries to solve the recursion equations it is implicitly handed to obtain closed solutions for the values of quantities changed. These solutions are represented as functions of the (symbolic) cycle index. The analyzer also produces an expresson for the number of cycles which will be taken. The "solutions" it derives are then handed back to the basic evaluator so that it can continue the evaluation.

3. Simplifier

The analyzer generates and manipulates various symbolic expressions. In order to facilitate the pattern matching techniques which it employs, we first translate the symbolic expression into a normal form. Normal form includes n-ary representation of addition and multiplication (the resulting arithmetic expressions being then simplified, employing the usual algebraic laws). Boolean expressions are put into a disjunctive normal form and simplified, again employing the usual boolean laws. The theorem proving ability of the present system comes from the ability to simplify a boolean expression representing the negation of a theorem to FALSE. The power of the boolean expression simplifier derives primarily from the incorporation of a complete solver for linear inequalities over the integers and reals.

IV. STATUS

At the present time simplified versions of the basic evaluator and simplifier are actually operational; a basic evaluator for full EL1 is expected by the end of August 1976. Some very preliminary components of the loop analyzer have been implemented and we anticipate a version which can handle loops with array manipulation and simple list processing by early October 1976.

Other work on symbolic evaluation is described in [1, 4, 5, 6, 7, 8].

BIBLIOGRAPHY

1. Boyer, Robert S., Elspas, Bernard, and Levitt, Karl N. SELECT - A Formal System for Testing and Debugging Programs By Symbolic Execution. Proc. International Conference on Reliable Software, 21-23 April 1975.

2. ECL Programmer's Manual. Center for Research in Computing Technology. TR23-74, 1974.

3. Cheatham, T. E., Jr., and Townley, Judy A. A Look at Programming and Programming Systems. Center for Research in Computing Technology, TR 18-75, 1975.

4. Deutsch, L. P. An Interactive Program Verifier. Ph.D. Thesis, Department of Computer Science, U. of California, Berkeley, June 1973.

5. Katz, Shmuel, and Manna, Zohar. Logical Analysis of Programs. Communications of the ACM, 19, 4 (April 1976), 188-206.

6. King, J. A Program Verifier. Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon U., Pittsburgh, Penn., 1969.

7. King, James C. Proving Programs to be Correct. IEEE Transactions on Computers, C-20, 11 (November 1971) 1331-1336.

8. Ruth, Gregory R. Analysis of Algorithm Implementations. Massachusetts Institute of Technology, Project MAC, MAC TR-130, May 1974.

Qualitative Analysis of Mathematical Expressions Using Computer Symbolic Mathematics

David R. Stoutemyer
Electrical Engineering
University of Hawaii

ABSTRACT

Increasingly powerful computers and simplification algorithms permit us to obtain answers for increasingly complex computer-algebra problems. Consequently, we will continue to get results which often are incomprehensibly lengthy and complicated. However, a user of computer-algebra systems need not abandon hope when faced with such results. Often the user is interested in *qualitative* properties of a result rather than details of an analytical representation of the result. For example, is the result real? bounded? even? continuous? positive? monotonic? differentiable? or convex? Where, if any, are the singularities, zeros, and extrema? What are their orders? What is the local behavior in the neighborhood of these notable features, as exhibited perhaps by series expansions? Are there simple asymptotic representations as certain variables approach infinity?

This paper describes a program which automatically analyzes expressions for some of these properties. The user may enquire about a specific property, such as monotonicity, or he may simply invoke a single function which attempts to determine all of the properties addressed by the collection of more specific functions. The specific functions are appropriate when a user knows which properties are important for his application, but frequently he is ignorant of the most decisive questions or ignorant of specific available functions which automatically investigate the desired properties. The collective qualitative analysis function is intended as a sort of *panic button*, which hopefully will provide some pleasantly surprising results that serve as a point of departure for further analysis. This function is a tool for deciphering unwieldy expressions that otherwise defy understanding.

Many of the above qualitative properties have numerous testable characterizations, and only a few have been explored here. However, the results of this initial effort indicate that qualitative analysis programs are a promising means of extending the utility of computer algebra.

This work was supported by National Science Foundation grant MCS75-22893 and grants to the Massachusetts Institute of Technology MATHLAB Project from the Advanced Research Projects Agency (ARPA), under Office of Naval Research Contract N00014-70-A-0362-0006.

KEY WORDS AND PHRASES: interval analysis, monotonicity, convexity, symmetry, periodicity, zeros, singularities, stationary points, qualitative analysis, symbolic algebraic manipulation, MACSYMA

CR CATEGORIES: 5.0, 5.4, 5.7

1. INTRODUCTION

Hamming [1962] has said "The purpose of computing is insight, not numbers", and computer symbolic mathematics helps fulfill that purpose by providing analytical rather than numerical answers. However, contemplation of machine-generated output expressions such as the following suggests that we generalize Hamming's statement by replacing "numbers" with "symbols":

```
                          4              7 X          1
   7 ERF(3 SINH(6 X - -) + 3 ASINH(--- + 9) - -)
                          9              3            2
   ---------------------------------------------
                          5

                          4              7 X
        5 SINH(6 X - -)   5 ASINH(--- + 9)
                          9              3
  7 TANH(--------------- + ---------------- - 2)
                 3                 3
+ ---------------------------------------------
                          5

                          4              7 X
        2 SINH(6 X - -)   2 ASINH(--- + 9)
                          9              3
  7 ATAN(--------------- + ---------------- - 8)
                 3                 3
+ ---------------------------------------------
                          5

                        1            5 X
+ 5 SINH(6 ERF(3 X - -) + 6 TANH(--- - 2)
                        2            3

                                                  1
                                        7 ERF(3 X - -)
        2 X          4                            2
+ 6 ATAN(--- - 8) - -)/9 + 5 ASINH(--------------
         3           9                     3

        5 X                2 X
  7 TANH(--- - 2)   7 ATAN(--- - 8)
         3                  3
+ --------------- + --------------- + 9)/9
         3                  3

                3    4 X - 3/7
  3 COSH(2 P + -) %E
                2
+ ---------------------------
                5
```

```
          3 P
     SQRT(--- + 2) + 5
          4                 4 X - 3/7        4 X - 3/7
  3 2                     %E              9 %E
+ ------------------------------- + -------------
                 5                             5

                 1                 3
  3 SQRT(2 P + -) COSH(2 P + -)
                 2                 2
+ ----------------------------
                 5

        3 P                     3                     3
  3 LOG(--- + 1) COSH(2 P + -)    6 COSH(2 P + -)
        5                     2                     2
+ ---------------------------- + ---------------
                 5                          5

                                                 3 P
                                            SQRT(--- + 2) + 5
                 1                     1          4
  9 SQRT(2 P + -)    3 SQRT(2 P + -) 2
                 2                     2
+ --------------- + ---------------------------------
        5                              5

          3 P
     SQRT(--- + 2) + 5
          4
  6 2
+ -------------------
          5

          3 P
     SQRT(--- + 2) + 5
          4                     3 P                 3 P
  3 2                     LOG(--- + 1)    9 LOG(--- + 1)
                                5                   5
+ ------------------------------- + -------------
                 5                             5

  18
+ --
  5
```

For computer-algebra output, the threshold of incomprehensibility is a matter of personal taste, temperament, and determination, but I conjecture that an expression exceeding one page is likely to also exceed the patience of most engineers and scientists. The threshold is dependent upon attractive output formatting such as tastefully located carriage returns, raised exponents, lowered subscripts, the use of braces and brackets as well as parentheses, and the suppression of operators or parentheses that can be implied. The use of short identifiers and the renaming of common subexpressions can also affect the threshold. Nevertheless, computer-algebra systems can easily generate results for which concise or regular-appearing representations do not exist--at least within the framework of the mathematical notation in which the problem is posed.

Perhaps the main reason that we cherish simple results is that they convey vital *qualitative* information. If so, we would be considerably less disappointed with an unwieldy result if this qualitative information could be automatically extracted from it. When we contemplate a simple expression we often immediately notice, or ought to notice, qualitative properties such as periodicity, oddness, evenness, positivity, singularities, or asymptotes. Such properties are often more important than either the details of an analytical representation or the numerical values that may be evaluated from the representation for specific numerical values of its indeterminates. We are most likely to notice these properties when an expression is comprised of not more than a few compositions of operators and functions with properties known to us, or when a lengthier composition of such operators and functions exhibits regular patterns. For example:

(a) When we see a sinusoidal subexpression we should think of its bounded periodic nature.
(b) When we see a logarithmic subexpression, we should think of its singularity and its monotonic-increasing concavity for positive arguments.
(c) When we see a sum of even, increasing, or convex subexpressions, we should think of the closure of these properties under summation.
(d) When we notice that the coefficients of terms are simply related to those of the binomial expansion or some well-known series expansion, it may suggest a more compact representation.

Qualitative properties such as these generally have testable characterizations which can be performed automatically using computer algebra. An appropriate analysis function can reduce an unmanageable expression to a single word such as "positive", "nonincreasing", or "continuous". Thus, qualitative analysis is a form of *simplification*--one which is particularly effective in reducing the *bulk* of an expression. It is somewhat analogous to replacing a lengthy set of numbers by a few of their statistics such as their mean and variance; in both situations we are willing to discard detailed information in order to expose the essence.

At least one computer-algebra system, MACSYMA, already contains some built-in functions that are useful for qualitative analysis, and this paper describes some additional functions that have been written to supplement them and to collect them all together into an easily-used package.

Section 2 briefly demonstrates the program on a few elementary examples, and Section 3 describes the underlying techniques used to implement the program. Section 4 summarizes the results for some more ambitious examples, with conclusions in Section 5.

2. A DEMONSTRATION

Some qualitative analysis functions have been written in the computer-algebra language MACSYMA described by the Mathlab group [1975]. Totaling about ten typewritten pages, the functions are too lengthy to list here, but copies are available from the author.

MACSYMA is interactive, prompting the user for each interaction cycle with a unique label beginning with the letter C. The user then types an expression terminated by a semicolon or a dollar sign. If terminated by the former, MACSYMA will generate and print a corresponding simplified expression with a corresponding label beginning with the letter D. There may also be intermediate printout, either unlabeled or labeled beginning with the letter E. %PI denotes π and %E denotes the base of the natural logarithms.

Suppose, then, that we wish to analyze the following expression:

```
(C3) TANH(COS(X/3)**2-COS(X/3-%PI/2)**2) +
     SIN(X/5)*ERF(TAN(X)) + ATAN(Y);
TIME= 81 MSEC.
                  X                                2 X
(D3) ATAN(Y) + SIN(-) ERF(TAN(X)) - TANH(SIN (-))
                  5                                3

                                                      2 X
                                                  - COS (-))
                                                      3
```

To do a qualitative analysis of this expression:

```
(C4) QUAL(D3);
                    LEARN(Y >= MINF)

                    LEARN(Y <= INF)

(E4)        BOUNDS = [- 3.3323905, 3.3323905]

(E5)                SLOPE(X) = UNKNOWN

(E6)              SLOPE(Y) = NONDECREASING

(E7)                CURVATURE = UNKNOWN

(E8)                SYMMETRIES(X) = EVEN

(E9)              SYMMETRIES(Y) = UNKNOWN

(E10)               PERIOD(X) = 30 %PI

(E11)                PERIOD(Y) = INF

                                 X
(E12) ZEROS = [ATAN(Y) + SIN(-) ERF(TAN(X))
                                 5

                                      2 X       2 X
                              - TANH(SIN (-) - COS (-)) = 0]
                                      3         3

(E13)               SINGULARITIES = []

                    LEARN(X >= MINF)

(E14)             LIMITAS(X, MINF) = UND

                    LEARN(X <= INF)

(E15)             LIMITAS(X, INF) = UND

                                 X
(E16) LIMITAS(Y, MINF) = SIN(-) ERF(TAN(X))
                                 5

                                      2 X       2 X     %PI
                              - TANH(SIN (-) - COS (-)) - ---
                                      3         3       2

                                 X
(E17) LIMITAS(Y, INF) = SIN(-) ERF(TAN(X))
                                 5

                                      2 X       2 X     %PI
                              - TANH(SIN (-) - COS (-)) + ---
                                      3         3       2

(E18)           NO STATIONARY POINTS FOUND
TIME= 13789 MSEC.

(D18) [E4, [E6, E5], [E7], [E8, E9], [E10, E11],

                [E12, E13], [E14, E15, E16, E17], [E18]]
```

QUAL has revealed that our expression is bounded and nondecreasing in y while even and periodic in x.

MSEC represents computing time in milliseconds for the MACSYMA-consortium PDP-10 KL10 computer. MINF represents $-\infty$, and INF represents ∞. The lines beginning with "LEARN" inform us of default bounds that have been established for the indeterminates in our expression. We may use the LEARN function to establish bounds of our own choosing. For example:

```
(C19) LEARN(P>0, Q>=-1/4, Q<=1/4, E<=EMAX);
TIME= 15 MSEC.
```

We may also individually invoke the various property-analysis functions invoked by QUAL. For example:

```
(C20) -ASIN(Q+1/2) - Q**2 + P*Q - 4*Q
      + LOG(P) - P**2 + 3*P + SQRT(P) - B**2 + B - 18;
TIME= 28 MSEC.
                  1     2                                  2
(D20) - ASIN(Q + -) - Q  + P Q - 4 Q + LOG(P) - P  + 3 P
                  2

                                                        2
                                    + SQRT(P) - B  + B - 18
(C21) CURVATURE(D20);
                    LEARN(P <= INF)

TIME= 4930 MSEC.
(D21)                 STRICTCONCAVE

(C22) EXPAND((X-1)**4 + (X-1)**2 + (Y-2)**2 + 5);
TIME= 62 MSEC.
       2          4      3      2
(D22)  Y  - 4 Y + X  - 4 X  + 7 X  - 6 X + 11

(C23) STATIONARYPOINTS(D22);
(E23)      STATIONARY POINTS = [[X = 1, Y = 2]]

(E24)     CORRESPONDING EXPRESSION VALUES = [5]

(E25)         CORRESPONDING TYPES = [MINIMUM]
TIME= 1424 MSEC.
(D25)                  [E23, E24, E25]

(C26) EXPAND((LOG(X-3)-5)*(X-Y+8)/TAN(X-Y+2));
TIME= 85 MSEC.
       LOG(X - 3) Y          5 Y           X LOG(X - 3)
(D26) ------------ - ------------ - --------------
      TAN(Y - X - 2)  TAN(Y - X - 2)  TAN(Y - X - 2)

        8 LOG(X - 3)         5 X              40
      - ------------ + ------------ + ------------
        TAN(Y - X - 2)  TAN(Y - X - 2)  TAN(Y - X - 2)

(C27) ZEROSANDSINGULARITIES(D26);
(E27) ZEROS = [COT(Y - X - 2) = 0, Y - X - 8 = 0,

                                          LOG(X - 3) - 5 = 0]

(E28) SINGULARITIES = [Y - X - %PI INTEGER - 2 = 0,

TIME= 1678 MSEC.                                  X - 3 = 0]

(D28)                  [E27, E28]
```

QUAL also automatically reveals successive levels of the structure of lengthier expressions. For example, the second derivative of D26 with respect to X is about 2 pages long; but the following analysis reveals that it is the sum of 21 terms, indicating the number of factors in each term:

```
(C29) QUAL(DIFF(D26,X,2))$
                REVEAL(..., 1) = SUM(21)

REVEAL(..., 2) = PRODUCT(5) + PRODUCT(4) + PRODUCT(5)

 + PRODUCT(4) + PRODUCT(4) + PRODUCT(3) + PRODUCT(4)

 + PRODUCT(3) + PRODUCT(3) + PRODUCT(3) + PRODUCT(5)

 + PRODUCT(4) + PRODUCT(5) + PRODUCT(4) + PRODUCT(3)

 + PRODUCT(5) + PRODUCT(4) + PRODUCT(5) + PRODUCT(4)

 + PRODUCT(4) + PRODUCT(3)

(E29)              BOUNDS = [MINF, INF]

(E30)               SLOPE(X) = UNKNOWN

(E31)               SLOPE(Y) = UNKNOWN

(E32)             CURVATURE = UNKNOWN

QUIT
```

The computation was manually interrupted at this point, because it had consumed a total of 60 seconds--mostly working on SECONDDERIVS. For non-trivial expressions, it often happens that some of the property-analysis functions require an inordi-

nate amount of time or space while others require only modest resources. Thus, access to the individual property-analysis functions is indispensible; and we must expect to be disappointed some of the time. This qualitative analysis package is no panacea. Although alternative algorithms and implementations could undoubtedly substantially increase the efficiency and effectiveness, there will always be expressions that are too complicated to analyze economically.

3. TECHNIQUES

Each of the subsequent subsections discusses the techniques used by one of the second-level functions invoked by the top-level function QUAL. Each of these second-level functions is directly usable too.

3.1 REVELATION

MACSYMA has a built-in function which may be used in the form REVEAL (*expression,level*). It reveals the form of its first argument to the operator level indicated by its second argument. As a trivial example, with u having the value $3*x + 8*sin(x)*x\uparrow 3$, REVEAL ($u$,1) yields SUM (2) and REVEAL (u,2) yields PRODUCT (2) + PRODUCT (3). Provided the given function exceeds a minimum complexity, the function REVELATION uses REVEAL to automatically display successive levels of complexity, while the output of REVEAL does not exceed another given level of complexity. Direct usage is of the form REVELATION (*expression,minimum,maximum*). Omitted trailing arguments have defaults: 200 for the second and 300 for the third. No printout occurs if *minimum* exceeds the number of print characters of *expression*, and printout ceases when the output of REVEAL exceeds *maximum* print characters or is identical to the previous level.

3.2 BOUNDS

We are often interested in knowing an upper bound and/or a lower bound for an expression. For a great variety of purposes, much of mathematical analysis is devoted to deriving such bounds.

To compute a *tight* bound--a least upper bound or a greatest lower bound--we must determine the supremum or infimum of an expression over its domain. These are general constrained optimization problems, and they may be attacked by general-purpose symbolic optimization programs. Stoutemyer [1975] describes one such program which is available on the same computer as this qualitative analysis program. However, that program is unlikely to succeed for the complicated expressions that necessitate a qualitative analysis program; and if it did succeed, the results would probably be more complicated than the original expression, which is incompatible with the spirit of qualitative analysis. Consequently, the BOUNDS function uses another technique called interval analysis, which is capable of treating far more complicated expressions. Interval analysis often gives loose bounds when an indeterminate occurs more than once in an expression, but loose bounds are often adequate for qualitative analysis. For example, it is often sufficient to know that a lower bound is positive or that an upper bound is finite. If a user is disappointed with a bound returned by BOUNDS, he may always try the above-mentioned optimization program. Also, there are relevant built-in IS and SIGN functions written by Michael Genesereth, as described by the Mathlab group [1975].

Interval analysis is thoroughly described in Moore [1966], Hanson [1969], and Nickel [1975]. Although originally intended for floating-point error analysis, interval analysis can be extended to symbolic expressions containing indeterminates with either numerical or symbolic bounds.

The function BOUNDS(*expression*) works as follows: If *expression* is a number, BOUNDS returns that number as both bounds. If *expression* is an indeterminate, BOUNDS retrieves the bounds from the database established by the LEARN function, establishing appropriate default values of MINF or INF where bounds have not previously been established. Currently, BOUNDS assumes that bounds on relevant indeterminates are of the form *item relation item* where *relation* is <, <=, >=, or >, one of the two sides is an indeterminate, and the other side is free of all indeterminates in *expression*. For strict inequalities, BOUNDS returns bounds of the form STRICT(*expression*).

If the leading operator of *expression* is "+", BOUNDS returns the sums of the respective bounds of the addends, which are obtained by recursive invocation of BOUNDS. Care is taken to properly combine bounds involving MINF, INF, or STRICT. To save time, subsequent addends are ignored whenever the cumulative bounds become [MINF,INF].

The minus operator does not occur in the general internal representation of MACSYMA, because it is absorbed into a numerical coefficient.

If the leading operator of *expression* is "*", BOUNDS cumulatively combines the bounds of each factor with the bounds of the partial product of preceeding factors, using recursion to obtain the bounds of each factor. Let $[\underline{u},\bar{u}]$ denote the bounds of a partial product u and $[\underline{v},\bar{v}]$ denote the bounds of the next factor v. In general, the bounds of uv are given by

$$[\min(\underline{uv},\underline{u}\bar{v},\bar{u}\underline{v},\bar{u}\bar{v}),\ \max(\underline{uv},\underline{u}\bar{v},\bar{u}\underline{v},\bar{u}\bar{v})], \qquad (1)$$

and MACSYMA does have built-in MIN and MAX functions. Unfortunately, if any of the constituent bounds are symbolic, MINF, INF, or involve STRICT, then a straightforward use of (1) results in an answer that is considerably more complicated than necessary--often more complicated than the original expression, which is incompatible with the purpose of qualitative analysis. Provided a global variable named PREDERROR is set to FALSE, MIN and MAX are capable of using the data-base established by LEARN to perform some simplification for symbolic arguments. However, we may do better, taking advantage of implicit relations such as the fact that lower bounds never exceed their companion upper bound.

When the sign of $\underline{u}$, $\bar{u}$, $\underline{v}$, or $\bar{v}$ is known, we may use a simpler form than (1). Figure 1 portrays the major essentially distinct possibilities with respect to the signs of these constituents. Many of

these cases may be treated together by interchanging the role of u and v and/or making use of relations such $uv=-[(-u)v]$ or $uv=[(-u)(-v)]$. Each rectangle indicates a typical domain of uv with a particular choice of signs for $\underline{u}$, $\overline{u}$, $\underline{v}$, and $\overline{v}$. The domain includes any edge not corresponding to a strict bound. Possible sites of the lower bound and upper bound of uv are respectively indicated by a small circle and an ×. For example, suppose that $\underline{u}$ and $\underline{v}$ are positive. Then, even if $\underline{u}$ and/or $\overline{v}$ are symbolic, we know that the bounds of uv are $[\underline{uv},\overline{uv}]$. As another example, suppose that $\underline{u}$ and $\overline{v}$ are positive, while $\underline{v}$ is symbolic. Then the domain could be the rectangle in the first quadrant or the one spanning the first and fourth quadrants. Consequently, the bounds are

$$[\min(\underline{uv},\overline{uv}),\ \overline{uv}] \ . \qquad (2)$$

As with "+", care is taken to properly combine bounds involving MINF, INF, or STRICT. Experience indicates that it is well worth the effort to avoid precludable arguments to MAX and MIN. Here, as elsewhere in computer algebra, effective simplification is aided by avoiding complexity in the first place.

Division does not occur in the general internal representation of MACSYMA, because u/v is represented as uv^{-1}.

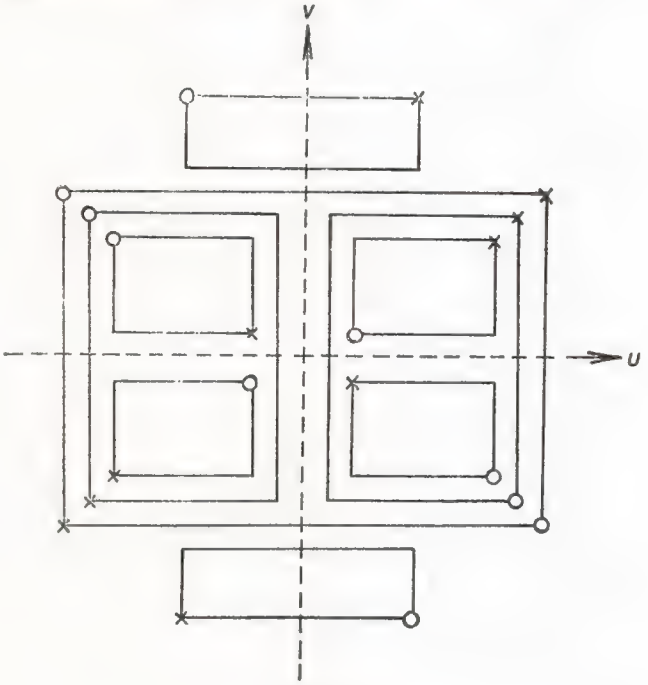


FIGURE 1
DOMAIN TYPES FOR THE INTERVAL OF UV

If the argument of BOUNDS is of the form u^v then the program first recurses to determine the bounds of u and v. Figure 2 portrays the essentially distinct domains that may result for u^v, using notation similar to Figure 1. Many of these cases may be treated together by using identities such as $u^v=1/u^{-v}=(1/u)^{-v}=1/(1/u)^v$. Noninteger powers of a negative number do not comply with the implicit assumption that we are dealing with real quantities throughout; so BOUNDS(u^v) returns [INFINITY,INFINITY] if the lower bound of u is negative or symbolic while the bounds of v are noninteger, where INFINITY represents complex ∞. In this context, only integer bounds on v are assumed to imply that v may assume only integer values.

If the top-level operator of the argument of BOUNDS is a built-in monotonic function, such as ATAN, TANH, or SINH, then BOUNDS uses the built-in LIMIT function to evaluate the limit of the function at each of the bounds of its arguments. Care is taken to properly treat STRICT as well as MINF and INF. Nonmonotonic functions are more awkward to treat, but some of them, such as SIN, COS, SECH, or COSH, have at least one finite bound even for arguments with bounds [MINF,INF].

Since interval analysis yields results which are dependent upon the form of an expression, the user may experiment with alternative forms if disappointed with the bounds on the initial form. For example, he could try expanded, factored, nested, partial fraction, continued fraction, or hybrid forms.

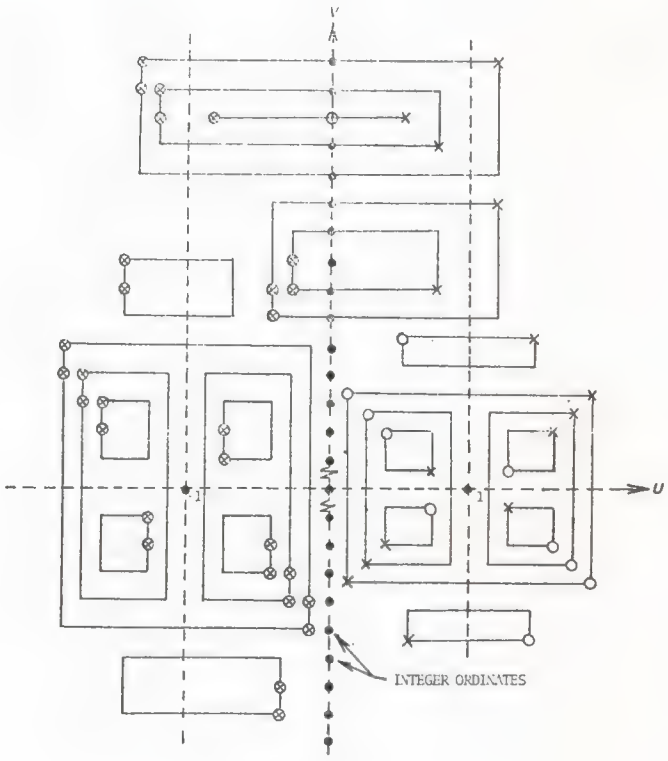


Figure 2
DOMAIN TYPES FOR THE INTERVAL U^V

3.3 SLOPES

The function SLOPES(*expression*,*indeterminates*) returns a list of E-labels of equations of the form SLOPE(*indeterminate*)=*type*, one for each indeterminate in *indeterminates*, where *type* is DECREASING, NONINCREASING, CONSTANT, NONDECREASING, INCREASING, or UNKNOWN, according to how the derivative of *expression* with respect to *indeterminates* is determined to behave in the domain established by previous LEARN commands. *indeterminates* is an indeterminate or a list of indeterminates. If omitted, it defaults to all of the indeterminates in *expression*. SLOPES simply uses BOUNDS on the first

derivatives, so it is not applicable to expressions containing nondifferentiable functions such as ABS, the built-in absolute-value function. Also, SLOPES may not return the strongest possible answer because of the interval analysis used by BOUNDS. For example, SLOPES may indicate NONDECREASING when an expression is actually increasing, or it may indicate UNKNOWN when an expression is actually polytonic. SLOPES is intended to indicate the monotonicity of differentiable functions, and the function described in Section 3.7 helps determine the domain of differentiability.

There are characterizations of monotonicity which do not require differentiability, but given the availability of BOUNDS, none is so easy to implement.

3.4 CURVATURE

The qualitative analysis program was first motivated by research in symbolic optimization, where it is important to determine if a domain is convex and if a function is concave or convex on that domain. However, it soon became apparent that most characterizations of convexity involve other qualitative properties such as monotonicity or bounds, and that they in turn involve other optimization problems. Moreover, most other qualitative properties such as symmetry, periodicity, singularities, and asymptotic behavior are also relevant to optimization. However, all of these interrelated properties are of interest throughout applied mathematics; so details of their specific use for optimization have been postponed for a later paper.

The function CURVATURE(*expression*) returns STRICTCONCAVE, CONCAVE, NONCONVEX, CONCAVEANDCONVEX, NONCONCAVE, CONVEX, STRICTCONVEX, NEITHERCONCAVENORCONVEX, or UNKNOWN, according to how a twice-differentiable *expression* is determined to behave within the domain implied by previous LEARN commands for its variables. The domain is necessarily convex because of the current restrictions on the bounds of indeterminates discussed in Section 3.2. However, for more general domains defined by the intersection of inequalities of the form $g(x_1,x_2,\ldots)>0$ and/or $g(x_1,x_2,\ldots)\geq 0$, it is sufficient that each g be convex on any convex domain containing the intersection of g with the more general domain. Consequently, CURVATURE could also be used to investigate the convexity of domains.

Although there are other characterizations of convexity, some of which do not require as much differentiability, one of the most straightforward characterizations is in terms of the definiteness of the Hessian matrix of second derivatives over the domain. An expression is strictly-concave, concave, concave-and-convex, strictly-convex, or neither-concave-nor-convex if the Hessian is respectively negative definite, negative semi-definite, rank-0, positive semi-definite, positive-definite, or indefinite throughout the domain.

There are, in turn, many characterizations of definiteness of a symmetric matrix. One of the most well-known is in terms of its eigenvalues. A symmetric matrix is negative definite, negative semi-definite, rank-0, positive semi-definite, positive definite or indefinite if the eigenvalues are all negative, all nonpositive, all zero, all nonnegative, all positive, or positive and negative, respectively. For matrices with symbolic elements, this requires finding the zeros of a characteristic polynomial with symbolic coefficients, then determining the signs of those zeros. This is forbidding unless the polynomial fortuitously factors into all linear factors over the integers.

A more practical test involves the signs of the determinants of the principal minors: For an n-by-n symmetric matrix, let $d_0=1$, d_j=the determinant of the leading j-by-j principal minor, (j=1,2,...n). Then the number of positive eigenvalues is the number of sign agreements between successive d_j, and the number of negative eigenvalues is the number of sign changes between successive d_j. Using diagonal pivoting to avoid or postpone zero determinants, any trailing zero determinants imply eigenvalues of value zero if there are no alternative nondiagonal pivots remaining. Otherwise, trailing zero determinants imply indefiniteness.

If Gaussian elimination is used to evaluate these determinants, they are simply the products of leading pivots. Consequently, the number of positive eigenvalues is simply the number of positive pivots and the number of negative eigenvalues is simply the number of negative pivots. Barth, Martin, and Wilkinson [1967] describes the use of this technique for iteratively determining the eigenvalues of symmetric tridiagonal matrices. Wilde and Beightler [1967] discuss the technique from the point of view of Langrange's technique for completing the square.

CURVATURE uses this technique, invoking BOUNDS to attempt determining the sign of each pivot, and terminating immediately if ever both a positive and negative pivot are discovered.

3.5 SYMMETRY

The function SYMMETRY(*expression*,*indeterminates*) returns a list of E-labels of equations of the form SYMMETRIES(*indeterminate*)=*type*, one for each *indeterminate* in *indeterminates*. The second argument and its default is as for SLOPES. *type* is EVEN, NONODD, ZERO, NONEVEN, ODD, NEITHER, UNKNOWN, PROBABLYEVEN, PROBABLYNONODD, NONODDANDPROBABLYNONEVEN, PROBABLYZERO, NONEVENANDPROBABLYNONODD, PROBABLYNONEVEN, or PROBABLYODD.
according to the determined symmetry of *expression* with respect to *indeterminate*, while all other indeterminates are arbitrary. The method is to substitute -*indeterminate* for *indeterminate* in *expression* to give $expression_2$, then to determine if *expression*-$expression_2$ and/or *expression*+$expression_2$ simplifies to zero or to a guaranteed nonzero using radical simplification and trigonometric simplification. If these simplified results are indecisive, a more powerful built-in function called ZEROEQUIV is used. Written by Joel Moses, ZEROEQUIV may occasionally give false indications for certain types of expressions. The manual clearly warns against using ZEROEQUIV on such expressions, but it is not yet clear how to automatically thoroughly check for them when ZEROEQUIV is used within another procedure. Consequently, any results found by ZEROEQUIV include the word

PROBABLY.

The user may also easily check for symmetry about some point other than the origin, by making an appropriate substitution. Good candidates are the zeros, singularities, and stationary points determined by the functions described in Sections 3.7 and 3.9.

3.6 PERIODS

The function PERIODS(*expression*,*indeterminates*) returns a list of E-labels of equations of the form PERIOD(*indeterminate*)=*value*, one for each indeterminate in *indeterminates*. *value* is 0 if *expression* is independent of *indeterminate* and INF if it is not periodic with respect to *indeterminate*. The second argument and its default is as for SLOPES. PERIODS first applies the built-in TRIGREDUCE function to combine products and powers of trigonometric functions into trigonometric functions of multiple arguments, thus driving trigonometric expressions towards explicit representation in terms of the shortest possible periods. Then, PERIOD determines if *indeterminate* occurs in *expression*, and if it occurs other than linearly as an argument of a trigonometric function. If *indeterminate* occurs only linearly as trigonometric arguments, the function determines the least common multiple of the periods for each occurance. Linearity and periods are determined by using the built-in RATCOEFF and FREEOF functions.

3.7 ZEROSANDSINGULARITIES

The function ZEROSANDSINGULARITIES(*expression*) returns a list of two equations. The first equation is of the form ZEROS=*equations* and the second is of the form SINGULARITIES=*equations*. Each equation in the list *equations* is of the form $expression_1$=$expression_2$. As indicated, either zeros or singularities occur at the solutions of these equations. Since they are generally multivariate, no attempt is made to solve any of these equations for one variable in terms of the others; but the user is free to do so, using the built-in SOLVE function.

The function simply factors the numerator and denominator, then extracts the arguments of any built-in functions that have singularities, such as LOG and TAN. Any factors of the numerator and singularities in the denominator that depend on any variable are taken as zeros of the expression, and vice-versa for singularities of the expression. Aside from initial cancellation of the greatest common divisor of the numerator and denominator, no attempt is made to cancel or combine zeros or singularities.

3.8 LIMITS

The function LIMITS(*expression*,*indeterminates*) returns a list of E-labels of equations of the form LIMITAS(*indeterminate*,*bound*)=*expression*. There is one such equation for each *bound* of each *indeterminate*, as established by default or by previous uses of LEARN. The second argument of LIMITS and its default are as for SLOPES.

LIMITS simply uses the built-in LIMIT function in an attempt to automatically reveal interesting asymptotic behavior.

3.9 STATIONARYPOINTS

The function STATIONARYPOINTS(*expression*,*indeterminates*) returns a list of the E-labels of

1. any stationary points of *expression* with respect to the indeterminates in *indeterminates*,
2. the corresponding values of *expression*,
3. the corresponding type -- MINIMUM, NONMAXIMUM, NONMINIMUM, MAXIMUM, SADDLE, or UNKNOWN.

The second argument and its default is as for SLOPES.

STATIONARYPOINTS simply equates the gradient of *expression* with respect to *indeterminates* to zero, solves the resulting equations using the built-in SOLVE function, substitutes any solutions into *expression*, and uses the technique of Section 3.4 to determine the definiteness of the Hessian matrix at each stationary point. For reasons discussed in Section 3.2, constraints implied by indeterminate bounds are ignored. If the user wishes otherwise, he may use the more general program referred to there.

4. TEST RESULTS

This section summarizes results for more difficult examples than the tutorial ones of Section 2. The examples in this section are intended to give a rough indication of the expression complexities that individual property-analysis functions are capable of analyzing in a reasonable amount of time and space. The threshold of unreasonable computation effort is, of course, a matter of individual taste and budgetary limitations. The times indicated below are for the MACSYMA-consortium PDP-10 KL-10 computer, using interpreted code, with a memory allocation level of 2 on a scale of 0 to 4.

BOUNDS required 24 seconds to return [STRICT(3.192-1427),INF] for the expression resulting from EXPAND(3*D6*D7 + 2*D9*D11/11), where P>0 and D6, D7, D9 and D11 are

$$(D6)\ Y^{4} + \%E^{TAN(5\,X - 6/5)} + SQRT\left(\frac{3\,P}{4} + 4\right) + LOG\left(\frac{2\,P}{3} + 1\right) + \frac{2}{9}$$

$$(D7)\ COSH\left(SEC\left(7\,X - \frac{8}{3}\right) - \frac{9}{2}\right) + \frac{4\,X^{E}}{3} + ACOSH\left(\frac{3\,P}{7} + 1\right) + \frac{3}{5}$$

$$(D9)\ -TANH\left(COTH\left(2\,X - \frac{9}{2}\right) - \frac{6}{7}\right) - SIN\left(\frac{3\,X}{5} - 7\right) + ERF\left(TAN\left(\frac{5\,X}{6} - 7\right) - 3\right) + \frac{1}{5}$$

$$(D11)\ COS\left(4\,X\,Y - \frac{8}{7}\right) - SECH\left(\frac{2\,Y}{7} - 3\right) - ATAN\left(CSC\left(6\,X - \frac{3}{4}\right) - 5\right) + \frac{1}{8}$$

Being about two pages long, there is insufficient space to list the expanded expression here. The interested reader is invited to generate it on his own computer-algebra system.

SLOPES required 36 seconds to return SLOPE(P)=INCREASING and SLOPE(X)=NONDECREASING for the expression listed in Section 1, with P positive. The expression was constructed to be increasing in X too, providing an example of how interval analysis may not lead to the strongest possible result.

CURVATURE required 12 seconds to return CONVEX for the expression below, with P positive:

$$\frac{3\,\%E^{4\,\cosh(Y+X+2)-3}}{2} - \frac{11\,Y}{3} - \frac{16\,X}{7}$$

$$+\ \frac{3\,\%E^{7\,\cosh(W)+3\,W^2-5\,W-11}}{5} + \frac{4\,W}{7} + \frac{3\,W^2}{4} - \frac{4\,W}{9}$$

$$-\ \frac{3\,\log\left(5\,\mathrm{atan}(P)^3+\frac{2}{2}\right)}{8} + \frac{2\,P^2}{5} - \frac{6\,P}{7}$$

$$-\ \frac{2\,\mathrm{sqrt}\left(3\,\tanh\left(\frac{2\,P}{5}+1\right)^3+\frac{7}{31}\right)}{9}$$

The expression is actually strictly convex, providing another example of non-tight bounds.

SYMMETRY required 4 seconds to return SYMMETRIES(X)=EVEN for the fourth derivative of the following expression:

$$\frac{\mathrm{atan}(X)\,\sin(X)+\%E^{\cos(X)}+X^2\,\cos(X)+X^2+7}{X^2+1}$$

Being about two pages long, this derivative is too long to list here. Again, the interested reader is invited to generate the expression on his own system.

PERIODS required 5 seconds to return 420π as the period of the expression

$$\frac{\mathrm{sqrt}\left(\sin\left(\frac{2\,2\,X}{5}\right)-\cos\left(\frac{2\,2\,X}{5}\right)+3\right)+\%E^{\cos(X/3)\,\sin(X/3)}}{\mathrm{atan}\left(2^{\sec(X/2)+3}\right)+\cot\left(\frac{X}{7}+2\right)}$$

For large expressions, factoring is liable to be the major expense in ZEROSANDSINGULARITIES, and the performance of the built-in MACSYMA factorization routine is described by Wang [1975]. Similarly, the built-in LIMIT function is likely to comprise the major expense for LIMITAS. The capabilities of LIMIT are described by Wang [1971].

For STATIONARYPOINTS, both solving the general nonlinear equations and determining the definiteness of the Hessian matrix can require an undue amount of computing resources. Stoutemyer [1975] explores the capabilities of a close-related program. However, in the context of qualitative analysis, limited experience suggests that for nontrivial cases where STATIONARYPOINTS succeeds, it often gives answers that are more complicated than the original expression. Thus, STATIONARYPOINTS is probably the least successful of the property-analysis functions in this package.

5. CONCLUSIONS

The algorithms and programs described here are exploratory ones, but these preliminary results suggest that:

(1) Qualitative analysis programs are a worthwhile component of general-purpose computer-algebra subroutine libraries.

(2) Automatic qualitative analysis is deserving of further research and development.

6. ACKNOWLEDGEMENTS

I am grateful to Carl Andersen, Richard Bogen, Rahul Chattergy, Richard Fateman, Michael Genesereth, Jeffery Golden, Jim Long, Tadaoshi Minimkawa, Dave Slepian, Evelyn Tasaka and the referees for their help and and suggestions.

REFERENCES

Barth, W., Martin, R.S. and Wilkinson, J.H., [1967], "Calculation of the Eigenvalues of a Symmetric Tridiagonal Matrix by the Method of Bisection," *Numer. Math 9*, pp. 386-393; also [1971] in *Handbook for Automatic Computation, Vol. II, Linear Algebra*, (Wilkinson, J.H. and Reinsch, C., editors), Springer-Verlag, pp. 249-256.

Hamming, R.W., [1962], *Numerical Methods for Scientists and Engineers*, Mc-Graw-Hill Book Co., New York.

Hanson, E., [1969], *Topics in Interval Analysis*, Oxford University Press, London.

Mathlab Group, [1975], *MACSYMA Reference Manual*, Project MAC, MIT, Cambridge, Mass.

Moore, R.E., [1966], *Interval Analysis*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Nickel, K., [1975], *Interval Mathematics*, Springer-Verlag, New York.

Stoutemyer, D.R., [1975], "Analytical Optimization Using Computer Algebraic Manipulation," *ACM Transactions on Mathematical Software 1*, No. 2, pp. 147-164.

Wang, P.S., [1971], "Evaluation of Definite Integrals by Symbolic Manipulation," *Project MAC Technical Report 92*, MIT, Cambridge, Mass.

Wang, P.S., [1975], "Factoring Multivariate Polynomials Over the Integers," *Math of Comp*, 29, No. 131, pp. 935-950.

Wilde, D.J. and Beightler, C.S., [1967], *Foundations of Optimization*, Prentice-Hall, Inc., Englewood Cliffs, N.J., pp. 81-87.

The Efficient Calculation of Symbolic Determinants

J. Smit
Department of Electrical Engineering
Twente University of Technology
Postbox 217
Enschede
The Netherlands

ABSTRACT

Early attempts to solve linear symbolic equations have shown that the traditional Gauss elimination method, the best for numerical calculations, gives common factors in the result due to the algorithm. This caused a fast degeneration of this method, which is now known to be the worst for symbolic calculations.
After a survey of several algorithms, the paper goes into the details of the Sparse Laplace Expansion (SLE) and the Sparse Nested Minors method (SNM). An analytical analysis of these methods is made under the assumption that the matrix is large and sparse, i.e. that all matrix elements are nonzero with a probability η.
It was this statistical model which gave clear indications how the SLE method performs and how it could be compared with the SNM method.
Instead of the presentation of one fixed implementation of the SLE algorithm, the paper shows how it evolves from the statistical model.
Abstract computing times could be compared, as it was possible to isolate the time consuming parts within the algorithms. Comparisons between predicted and measured performance show that a reasonable good insight in the algorithms is obtained, even for those cases where the statistical model is not directly applicable.
A comparison between the methods shows that SLE is preferable as long as the average number of nonzero matrix elements in each row (column) is $\leq e+1$.
The results of the implementations of several speed ups indicate that improvements (in orders of magnitude?) can still be expected. There is hope that the, adaptive, utilisation of one remaining degree of freedom, still available in the SLE method, will show its practical feasability in the Double SLE method, currently in development.

1. Several methods to calculate the symbolic value of a determinant.

Here we restrict ourselves to a short explanation and an example which shows the properties of the method when applied to a small matrix. A description of the algorithms for the methods discussed is given in |1|.

1.1 Gaussian elimination.

Calculate: $\begin{vmatrix} A_1 & B_1 & C_1 \\ A_2 & B_2 & C_2 \\ A_3 & B_3 & C_3 \end{vmatrix}$

Eliminate: A_2, A_3. Note: the elimination introduces A_1^2 as a factor which is omitted.

$$\begin{vmatrix} A_1 & B_1 & C_1 \\ 0 & \begin{vmatrix} A_1 & B_1 \\ A_2 & B_2 \end{vmatrix} & \begin{vmatrix} A_1 & C_1 \\ A_2 & C_2 \end{vmatrix} \\ 0 & \begin{vmatrix} A_1 & B_1 \\ A_3 & B_3 \end{vmatrix} & \begin{vmatrix} A_1 & C_1 \\ A_3 & C_3 \end{vmatrix} \end{vmatrix} = \begin{vmatrix} A_1 & B_1 & C_1 \\ 0 & (A_1B_2-A_2B_1) & (A_1C_2-A_2C_1) \\ 0 & (A_1B_3-A_3B_1) & (A_1C_3-A_3C_1) \end{vmatrix}$$

Eliminate: $\begin{vmatrix} A_1 & B_1 \\ A_3 & B_3 \end{vmatrix} = (A_1B_3-A_3B_1)$.

Note: the elimination introduces $\begin{vmatrix} A_1 & B_1 \\ A_2 & B_2 \end{vmatrix}$ as a factor, which is omitted again.

$$\begin{vmatrix} A_1 & B_1 & C_1 \\ 0 & \begin{vmatrix} A_1 & B_1 \\ A_2 & B_2 \end{vmatrix} & \begin{vmatrix} A_1 & C_1 \\ A_2 & C_2 \end{vmatrix} \\ 0 & 0 & \begin{vmatrix} \begin{vmatrix} A_1 & B_1 \\ A_2 & B_2 \end{vmatrix} & \begin{vmatrix} A_1 & C_1 \\ A_2 & C_2 \end{vmatrix} \\ \begin{vmatrix} A_1 & B_1 \\ A_3 & B_3 \end{vmatrix} & \begin{vmatrix} A_1 & C_1 \\ A_3 & C_3 \end{vmatrix} \end{vmatrix} \end{vmatrix}$$

Result:

$$\text{Det} = \frac{A_1 \begin{vmatrix} A_1 & B_1 \\ A_2 & B_2 \end{vmatrix} \begin{vmatrix} \begin{vmatrix} A_1 & B_1 \\ A_2 & B_2 \end{vmatrix} & \begin{vmatrix} A_1 & C_1 \\ A_2 & C_2 \end{vmatrix} \\ \begin{vmatrix} A_1 & B_1 \\ A_3 & B_3 \end{vmatrix} & \begin{vmatrix} A_1 & C_1 \\ A_3 & C_3 \end{vmatrix} \end{vmatrix}}{A_1^2 \begin{vmatrix} A_1 & B_1 \\ A_2 & B_2 \end{vmatrix}}$$

The calculation is well suited for floating point calculations, and can easily be extended to give the inverse of a matrix. Bareiss |2| has shown a modification on the above algorithm, a fraction free method, which is better than Gaussian elimination for integer constants and also for rational constants.

1.2 Fraction Free method.

Division of numerator and denumerator of the previous example by: $A_1 \begin{vmatrix} A_1 & B_1 \\ A_2 & B_2 \end{vmatrix}$ gives:

$$\text{Det} = \frac{\begin{vmatrix}(A_1B_2-A_2B_1)(A_1C_2-A_2C_1)\\(A_1B_3-A_3B_1)(A_1C_3-A_3C_1)\end{vmatrix}}{A_1}$$

The above divisions were trivial and can be accounted for in a general elimination method. Also A_1 divides exactly, however this division is far less trivial.

$$\text{Det} = \frac{(A_1B_2A_1C_3-A_1B_2A_3C_1-A_2B_1A_1C_3+A_2B_1A_3C_1)/A_1}{-(A_1B_3A_1C_2-A_1B_3A_2C_1-A_3B_1A_1C_2+A_3B_1A_2C_1)/A_1}$$

Note that A_1 cannot divide both sums. However if the sums are added then the terms $A_2B_1A_3C_1$ cancel, and all remaining terms have a factor A_1.

$$\text{Det} = A_1B_2C_3-A_1B_3C_2-B_2A_3C_1+B_3A_2C_1-A_2B_1C_3+A_3B_1C_2$$

The key point of the Fraction Free method is that a division which is known to be exact is undertaken. (See [2]). If the divisor (A_1) is a lengthy multivariate polynimial (expression) then the dividability implies that the divident necessarily has to be a much larger expression. This is one of the disadvantages of the Fraction Free method. (See also [1]). The method is however well suited for integer and rational calculations, as it provides a cheap mechanism to avoid excessive growth of numbers.

1.3 Nested minors.

This method calculates first all possible 2×2 minors from the two bottom rows, then all possible 3×3 minors from the three bottom rows, etc.

EXAMPLE:

$$\begin{vmatrix}A_1 & B_1 & C_1 & D_1\\A_2 & B_2 & C_2 & D_2\\A_3 & B_3 & C_3 & D_3\\A_4 & B_4 & C_4 & D_4\end{vmatrix} =$$

$$\begin{array}{l}+A_1(B_2(C_3D_4-D_3C_4)-C_2(B_3D_4-D_3B_4)+D_2(B_3C_4-C_3B_4))\\-B_2(A_2(C_3D_4-D_3C_4)-C_2(A_3D_4-D_3A_4)+D_2(A_3C_4-C_3A_4))\\+C_1(A_2(B_3D_4-D_3B_4)-B_2(A_3D_4-D_3A_4)+D_2(A_3B_4-B_3A_4))\\-D_1(A_2(B_3C_4-C_3B_4)-B_2(A_3C_4-C_3A_4)+C_2(A_3B_4-B_3A_4))\end{array}$$

Since the answer contains only 6 different 2×2 minors, the method saves 6 calculations of a 2×2 minor. The method is especially usefull if the matrix is *dense* (contains relatively few zero's) and its entries are relatively complicated expressions. (See [1,3]).

1.4 Laplace expansion.

This method calculates all terms of the determinant using:

$$|A| = \sum_{j=1}^{m} a_{1j}(-1)^{1+j}\,|M_{1j}|$$

See also 2.4.
For the example of 1.3 one obtains: $|A|$ =

$$\begin{array}{l}+A_1B_2C_3D_4-A_1B_2D_3C_4-A_1C_2B_3D_4+A_1C_2D_3B_4+A_1D_2B_3C_4-A_1D_2C_3B_4\\-B_1A_2C_3D_4+B_1A_2D_3C_4+B_1C_2A_3D_4-B_1C_2D_3A_4-B_1D_2A_3C_4+B_1D_2C_3A_4\\+C_1A_2B_3D_4-C_1A_2D_3C_4-C_1B_2A_3D_4+C_1B_2D_3A_4+C_1D_2A_3B_4-C_1D_2B_3A_4\\-D_1A_2B_3C_4+D_1A_2C_3B_4+D_1B_2A_3C_4-D_1B_2C_3A_4-D_1C_2A_3B_4+D_1C_2B_3A_4\end{array}$$

The disadvantage of this method is that in fact recalculation of identical minors is requested. It was implemented in NETFORM in 1970 as a sparse matrix method. (See for example [4]).

1.5 Evaluation - interpolation.

The previous examples show directly that the result of a determinant calculation is a multivariate polynomial. The *evaluation-interpolation* method is based on this fact. Suppose that the entries of the matrix are polynomials in one variable X. The determinant calculation results in a polynomial $P_n(X)$. This polynomial can be evaluated in n+1 basis points X_i by substitution of this value in the matrix entries. This gives a numerical problem which may be solved by means of the *Fraction Free* method. Interpolation of the $P_n(X_i)$ will finally result in $P_n(X)$. This method has been extended to several variables.

1.6 Characteristic polynomial.

This algorithm will not be discussed here, See [1,5].

1.7 Discussion.

E. Horowitz and S. Sahni discussed most of the above algorithms mentioned,very thoroughly in [1]. They defined the computing time as the required number of coefficient multiplications*).
The matrix entries are polynomials which may be *dense*, *sparse***) or such that an *average growth* model applies. They found the Nested Minors method to be best in both the *average growth* amd the *sparse case*. The method of Laplace Expansion is not covered in the article because it is worse in any case. This is true because they assume the matrix to be dense (even full). The next section will show how Laplace Expansion may advantageously be used if the matrix is *sparse*.

2. The calculation of the determinant of a sparse matrix.

2.1 Definition of a sparse matrix.

Since there is no general definition of a sparse matrix, one will be given here. It allows a statistical computation for the calculation of the expected number of terms generated by the Laplace Expansion method, and for the expected number of minors generated by the Nested Minors method.

Definition:

A matrix is, for the purpose of the analysis in this chapter, called sparse if:
1) $p\{a(i,j)\neq 0\}=\eta<<1$ for all i,j and consequently,
2) $p\{a(i,j)\neq 0 \mid a(k,1)\neq 0,\ i\neq k \text{ and or } j\neq 1\}=\eta$.

This definition implies that a tridiagonal matrix is a *very improbable* case of a sparse matrix (See the use in [3] of such a matrix as a test case for the Sparse Nested Minors method).

Further restrictions on η will be given below, in order to enhance the meaning of "*sparse matrix*". A comparison between the statistics of our model and an actual calculation will be implicitly given by means of the specification of the measured values.

*) Example: $(7x+3)+(4x^2+2x+13)=7x(4x^2+2x+13)+3(4x^2+2x+13)$ requires 6 coefficient multiplications.

**) Example: $4x^5+3x^4+7x^3+x^2+2x+7$ is *dense*, $4x^5+2x$ is *sparse*.

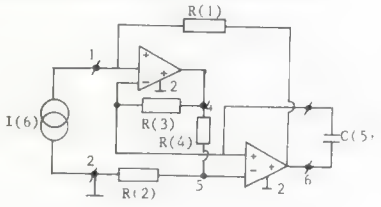
2.2 Introduction in the terminology using an example.

A matrix generated by the NETFORM system for the analysis of a gyrator circuit due to A. Antoniou [6] is taken as an example (fig. 2.2,1).
The terminology used is as follows:

entity	*symbol*	*value in example*
matrix	[A]	See fig. 2.2,2
matrix size (m × m)	m	20
level of recursion	r	1,...,20
probability that a(i,j)≠0	η	3/20
number of terms*) at recursion level r	τ(r)	-
- for sparse method at recursion level r	$\tau_s(r)$	See fig. 2.3,1
number of minors**) at recursion level r	μ(r)	See fig. 2.3,1
- for sparse method at recursion level r	$\mu_s(r)$	See fig. 2.3,3

Note the value of τ_s and μ_s are expected values. These values depend on the value of η.

The gyrator circuit of A. Antoniou.



The circuit.

```
BEGIN:
  * THIS IS A GYRATOR BY A ANTONIOU
  R (1,6);R (2,5);R (3,4);R (4,5);
  C (3,6);I (2,1);
  SUB (3,1,2,4);
   > OPAMP;
  END;
  SUB (3,5,2,6);
   > OPAMP;
  END;
  SOLVE U(#(6));CALC;
END;
```

The NETFORM description.

Fig. 2.2,1

*) τ(r) corresponds with the Laplace Expansion method. See 1.4.
$\tau_s(r)$ corresponds with the Sparse Laplace Expansion method. See 1.4.

**) μ(r) corresponds with the Nested Minors method. See 1.3.
$\mu_s(r)$ corresponds with the Sparse Nested Minors method. See 1.3.

NETFORM builds the following matrix for the determinant of the numerator of U((6)).

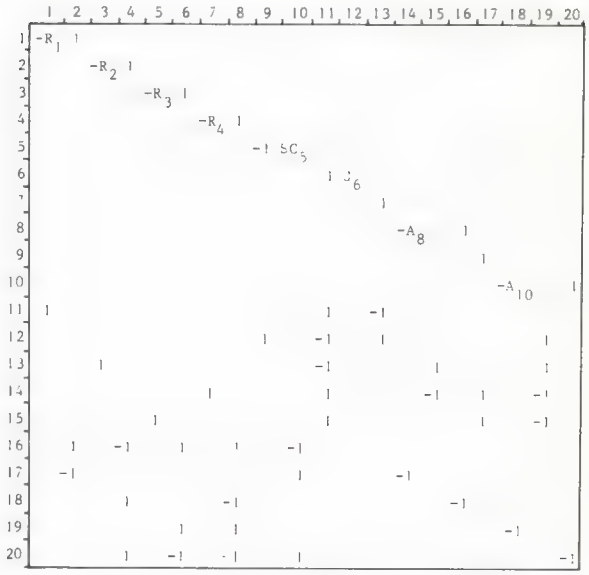


Fig. 2.2,2

The following methods are analyzed:

name	*abbreviation*
Laplace Expansion	LE
Sparse Laplace Expansion	SLE
Nested Minors	NM
Sparse Nested Minors	SNM

Remember that in this context the word *sparse* always refers to the sparsity of the matrix, not of the polynomials.

2.3 Storage and time estimations.

The LE and SLE algorithms spend most of their *time* in the generation of terms of the expansion. The total amount of *storage* needed is proportional to the number of terms actually generated (τ(m), $\tau_s(m)$). The NM and SNM algorithms spend most of their *time* in the generation (and simplification) of minors. Here the total amount of *storage* is proportional to the maximum number of minors (μ_{max}, μ_{smax}) generated. The example introduced in 2.2 will be analyzed in order to make comparisons between the methods possible.

LE: All terms at recursion level r occur with a probability 1 (η=1), where

$$\tau(r) = \binom{m}{r} r! = \frac{m!}{(m-r)!}$$

If $\tau_{max} = \max_{1 \le r \le m} |\tau(r)|$ then it is obvious that

In our example, for instance, we have $\tau_{max}=\tau(20)=20!=243.10^{16}$. This implies that the LE method can never be a serious competitor.

SLE: At recursion level r in the SLE development all terms ($\tau_s(r)$) occur with a probability η^r, so their expected number is:

$$\tau_s(r) = \binom{m}{r} r! \eta^r = \frac{m!}{(m-r)!} \eta^r$$

Only terms without any factor zero need to be converted into the respective symbolic form.

They can be found at r=m. The expected value of $\tau_s(m)$ is:

$$\tau_s(m) = m!\eta^m \sim \sqrt{2\pi m}\,(\frac{m\eta}{e})^m \quad *)$$

Our example gives: $\tau\ (20) = 243.10^{16}$,
$\tau_s(20) = 80.90$,
and $\tau\ (20)/\tau_s(20) \simeq 3.10^{16}$.

The intermediate expectation $\tau_s(r)$ is plotted in fig. 2.3,1. The area $\sum_r \tau_s(r)$ is 63,567 while 47,218 is actually counted in the example. Terms which are not known to be nonzero are stored on the recursion stack, which has a maximum depth: r=m. This shows that the amount of storage is proportional to $\tau_s(m)$ and that the amount of time needed is proportional to the area: $\sum_r \tau_s(r)$.

For relatively sparse matrices $\tau_{s\max}$ is relatively large compared with $\tau_s(m)$. This implies that the computing time needed to simplify the generated expression can be neglected in comparison with the time needed for the recursion. Note that $\tau_s(r)$ can also be written as:

$$\tau_s(r) = \prod_{j=1}^{r} \eta(m-j+1)$$

This expression shows the exponential nature of $\tau_s(r)$ (See fig. 2.3,2).

NM: The number of minors, $\mu(r)$, at recursion level r is equal to the number of ways in which r columns may be selected from a total of m columns. Hence

$$\mu(r) = \binom{m}{r} \qquad \text{(See fig. 2.3,1)}$$

Let

$$\mu_{max} = \max_{1\le r\le m} |\mu(r)|$$

It is obvious that μ_{max} is found for r=m/2 if m is even and for $r=m/2 \pm \frac{1}{2}$ if m is odd. Hence in our example, we find

$$\mu_{max} = \binom{20}{10} = 184{,}756$$

This value puts so many restrictions on storage and time that a better method: SNM is required.

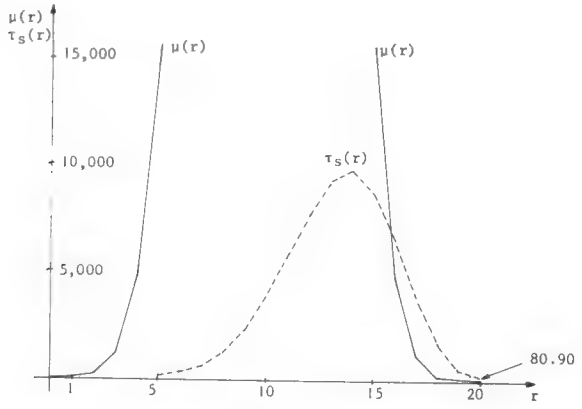
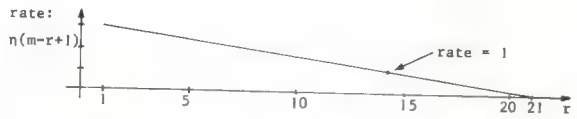


Fig. 2.3,1.



The rate of growth of $\tau_s(r)$ as a function of r.

Fig. 2.3,2.

SNM: If a distribution p(n) is known which gives the probability that n terms from the SLE development occur in one minor (from the NM development), then it is possible to calculate μ_s. *)

Let

$$\mathcal{E}(N) = \sum_{n=0}^{\infty} n\ p(n).$$

Then

$$\frac{\tau_s(r)}{\mu(r)} = \frac{\sum_{n=1}^{\infty} n\ p(n)}{\sum_{n=0}^{\infty} p(n)} = \mathcal{E}(N)$$

and

$$\frac{\mu_s(r)}{\mu(r)} = \frac{\sum_{n=1}^{\infty} p(n)}{\sum_{n=0}^{\infty} p(n)} = 1-p(0)$$

The remaining problem is to find such a distribution function. Since we are dealing with sparse matrices with $\eta<<1$, it is obvious that $\tau_s(r)=\eta^r\tau(r)<<\tau(r)$. Due to the sparsity even the mutual dependence between the occurrence of two terms vanishes. This implies that the terms are (almost) randomly distributed over the minors. This observation suggests that the geometric distribution might be a reasonable approximation.
The geometric distribution gives:

$$p(N=n)=(1-\alpha)\alpha^n,\ p(0)=1-\alpha,\ \mathcal{E}(N) = \frac{\alpha}{1-\alpha}$$

So

$$\frac{\tau_s(r)}{\mu(r)} = \frac{\alpha}{1-\alpha} \quad \text{and} \quad \frac{\hat{\mu}_s(r)}{\mu(r)} = \alpha$$

Elimination of α gives:

$$\hat{\mu}_s(r) = \frac{\mu(r)}{1+\frac{\mu(r)}{\tau_s(r)}} = \frac{\binom{m}{r}}{1+\frac{1}{\eta^r r!}}$$

$\hat{\mu}_s(r)$ is plotted in fig. 2.3,3.
Now a simple comparison is possible between $\mu_s(r)$, $\mu(r)$ and $\tau_s(r)$. Note that $\hat{\mu}_s(r) \simeq \tau_s(r)$ if $\mu>>\tau_s$ (r<12) and that $\hat{\mu}_s(r) \simeq \mu(t)$ if $\tau_s>>\mu$ (r>17).
The value of $\hat{\mu}_s(r)$ at r=m, $\hat{\mu}_s(m) = \frac{1}{1+\frac{1}{\eta^m m!}}$, 0.988 in the example, reflects the possibility that there may not be any term at all in the development. The difference between $\tau_s(r)$ and $\mu_s(r)$ is plotted in fig. 2.3,4.

*) The formula of stirling $n! \sim \sqrt{2\pi}\ n^{n+\frac{1}{2}}\ e^{-n}$ is used.

*) Attempts to calculate μ_s directly from the probability model failed due to the complicated nature of the problem.

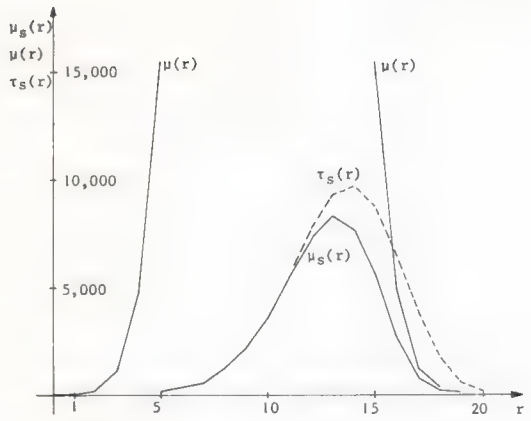


Fig. 2.3,3.

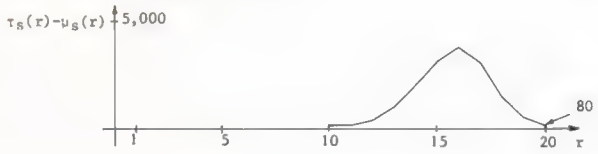


Fig. 2.3,4

The bottleneck of the SNM method is the linkage of new matrix elements with already found minors. Due to the fact that no space is available to store all possible $\binom{m}{r}$ minors, only all nonzero minors are stored in an ordered list. A bisection method will require a total amount of $\sim$ 13 steps, for the example to determine the location of a specific element in the list ($2^{13} = 8192$). This means that the SNM method may save ± 25% on counts, but the elementary step is about 13 times more complicated than the elementary step of the SLE method. In order to facilitate a comparison between the SLE and the SNM method, a factor X(= 13) can be introduced which accounts for this difference. (See also 2.3.1).

Conclusion:
In the example the SLE method is clearly preferable over the SNM method. This due to the sparsity of the matrix. An objective criterium to compare both methods can be found if the relation between η and m is considered.

2.3.1 Preference for NM, SNM and SLE depending on η and m.

As far as *storage* is concerned the SLE method is always preferable over the other methods, however for m<10 the storage problem of the NM method is no dominant factor.
Therefore for m<10 NM will be a good algorithm, as well as SLE if $\eta(m+1) \leq e+1$ as will be shown below.
Now the preference between SLE and SNM will be based on (not too precise) time comparisons. It is relatively easy to compare the time spent for SLE and SNM at the level of recursion at which SLE is most time consuming. At this level ($r_{max}=m+1-\frac{1}{\eta}$) SNM has a more complicated elementary step which is taken into account with a factor X $X=\log_2(\mu_s(r_{max}))$.

There will be a preference of SLE over SNM if:

$$\tau_s(r_{max}) \leq X\ \mu_s(r_{max})$$

or

$$\binom{m}{r_{max}}\ r_{max}!\ \eta^{r_{max}} \leq X\ \frac{\binom{m}{r_{max}}}{1+\frac{1}{\eta^{r_{max}}\ r_{max}!}}$$

This may be simplified to:

$$\eta^{r_{max}}\ r_{max}! = \frac{\tau_s(r_{max})}{\mu(r_{max})} \leq X-1$$

Using the formula of Stirling this may be written:

$$\sqrt{2\pi\ r_{max}}\ \left(\frac{\eta\ r_{max}}{e}\right)^{r_{max}} \leq X-1$$

or

$$\sqrt{2\pi\left(m+1-\frac{1}{\eta}\right)}\ \left(\frac{\eta(m+1)-1}{e}\right)^{m+1-\frac{1}{\eta}} \leq X-1$$

The dominant part in this inequality is:

$$\frac{\eta(m+1)-1}{e} \leq 1 \quad \text{or} \quad \eta(m+1) \leq e+1$$

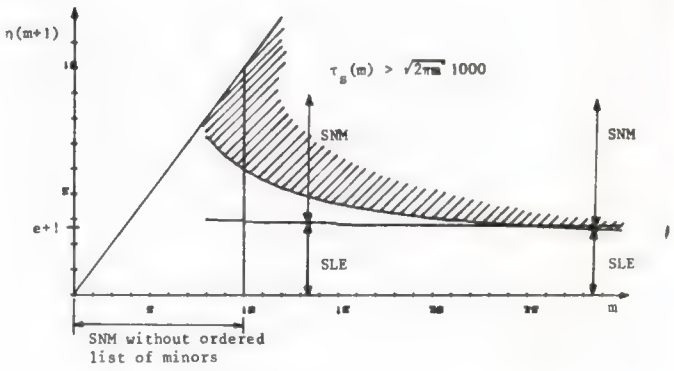


Fig. 2.3.1,1

The inequality gives the range in which the SLE method is *certainly* better than the SNM method. Note that the factor X introduced before plays no role in the comparison.
Fig. 2.3.1,1 shows the above results. The shaded area gives the region for which solutions are hard to obtain (and to interpret) due to the enormous number of terms in the solution. ($\tau_s(m) > \sqrt{2\pi m}\cdot 1000 \simeq 10.000$).

2.4 Sparse Laplace expansion in detail.

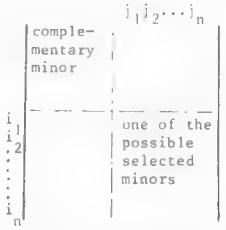
The preceding analysis of storage and time requirements justify a further analysis of the SLE method. The NETFORM implementation is, of course, based on the definition of the Laplace expansion. A global sketch of the role of the Laplace Expansion in the NETFORM system will be given.
Two major parts can be distinguised.

I. Handling the matrix (DENDET).
II. Calculation of the determinant (DENMIN).

Taking computing time into consideration, the search procedure, II.1 is the most important part of the algorithm. Therefore the construction of this part of the system will be judged on the basis of the definition of the Laplace Expansion. The handling of the matrix will not be described at all.

Definition:

Let any n rows (columns) be selected from an (m×m) matrix A. Then the determinant $|A|$ of A is equal to the sum of the products of all the n-th order minors contained in the chosen rows (columns) each multiplied by its algebraïc complement. The value of a 1-st order minor is equal to the value of its only matrix element a_{11}.



algebraïc complement=$(-1)^{(i_1+i_2+\ldots+i_n)+(j_1+j_2+\ldots+j_n)}$ * complementary minor.

Fig. 2.4,1.

If n=1 is taken then the definition gives:

$$|A| = \sum_{j=1}^{m} a_{1j}(-1)^{1+j}|M_{1j}| \qquad \text{(development along a row)}$$

or

$$|A| = \sum_{i=1}^{m} a_{i1}(-1)^{1+i}|M_{i1}| \qquad \text{(development along a column)}$$

The minor routine of NETFORM uses this recursion definition.

2.4.1 The search procedure.

The search procedure will attempt to find the nonzero terms in the development of the determinant. For this purpose the determinant calculation:

$$|A| = \sum_{j=1}^{m} a_{1j}(-1)^{1+j}|M_{1j}|$$

is split in:

1) The search procedure which recursively calculates all permutations of the trace (T), called for brevity minor sequences.

   $T(A) = a_{1j} \cdot T(M_{1j})$

   These minor sequences are composed of elementary matrix elements of distinct minors and correspond to (incomplete - if r<m) terms of the expansion, without appropriate sign.

2) The sign calculation mechanism determines the signs of all elementary matrix elements of the minor sequence only if the deepest level of recursion is reached (r=m).

3) The construction of the symbolic term t, using the minor sequence T(A) and the sign (σ) juist calculated:

   Hence: $t=\sigma \cdot T(A)$

   Note that the minor sequence is nothing else than the history of the recursion. It has to be converted into a product of symbols in order to represent a symbolic minor sequence. Finally it has to be added to the value the determinant, det

   det = det + t

   The discussion about a mixed symbolic numeric method [4,8] will be given in section 2.4.7.

The efficiency of the implementation of the search procedure depends on the representation of the matrix elements. In this context efficiency implies easy and rapid acces to the data structures, describing the matrix elements, as to obtain information about the possible presence of matrix elements. A list structure to be called the *skeleton* provides this information in a natural way:

1) To be able to develop along a row of A, the skeleton provides an along pointer connecting the nonzero matrix elements of a row.

2) To locate the minor a pointer to the first nonzero matrix element of the next row is provided.

Hence: the matrix elements of the minor are *implicitly* given by this method of development, which implies that the elements on the next rows in the skeleton are possible candidates. It remains to give an *explicit* indication to distinguish between the columns. The following approaches are possible:

1) Modify the skeleton such that the matrix elements on the skeleton are the matrix elements of the minor.
   N.B.: This method implies that at each recursion step (forwards or backwards) one column of the skeleton has to be modified.

2) A bit in a vector is set to indicate that a specific column is no longer part of the minor.
   N.B.: This method requires updating of the bit vector at each recursion step. Matrix elements in a row contribute to the minor if the bit is not set (the column is not occupied).

The second method is chosen in the NETFORM implementation; it is easy to implement. As a consequence a subscript is required to locate the position of the columns.

2.4.2 Sign calculations.

The actual calculation of the sign: $(-1)^{1+j}$ is deferred until a complete minor sequence is found. This saves a considerable amount of time, since such a sign computation is complicated. Note that the simple notation M_{1j} implies a renumbering of the subscripts in the minor in order to be able to apply the sign formula. A renumbering operation at each step of the search procedure would drastically degrade the search procedure, and is therefore omitted. Hence the value of j is counted afterwards for all matrix elements of the minor sequence. This procedure can slightly be improved if the result of a previous sign calculation is taken into consideration.

Examples of the search procedure and the sign calculation are given in [4,8].

2.4.3 The effect of the skeleton, expressed in properties of the minor.

At each new level of recursion the search procedure will try to find nonzero matrix elements at the next row belonging to the minor. If the top row of the minor is empty a return from this recursion step will be made.

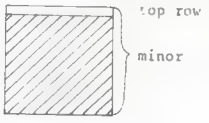


Fig. 2.4.3,1.

Note that this situation occurs particulary often if the top of the $\tau_s(r)$ curve is reached.*) If only one nonzero matrix element is available in the first row of the minor, then the search procedure will develop the minor with respect to this matrix element.

Fig. 2.4.3,2.

2.4.4 A speed up of the search procedure.

The adaptive measures taken to speed up the search procedure are considered first, directly followed by the static improvements. An entirely new algorithm will be proposed in 2.5.

I. One matrix element in a column.

This situation is easily detected by the search procedure if the element belongs to the top row of the minor; in other cases this is not detected as a special case.

Fig. 2.4.4,1

The presence of the matrix element after a return from the recursion may be neglected, since the search will continue along the row. This implies that the new minor will have an empty column. (See fig. 2.4.4,1). This situation may be easily recognized by the search procedure, because a bit in the skeleton can be used to indicate that this matrix element is the last in the column.

II. The down-subscript algorithm.

Another *simple* algorithm is implemented which adaptively speeds up the search procedure. It presumes that for each matric element in the skeleton the subscript of the matrix element below (i.e. in the same column) is given ("down subscript"). The algorithm is initiated at a recursion level say r. At this level the first row of the minor is supposed to be empty.

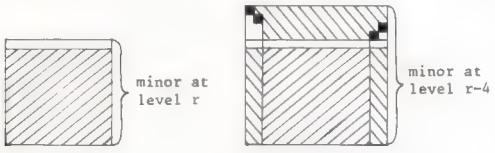


Fig. 2.4.4,2

At each return recursion the search procedure checks wheather the current matrix element is below the border found at level r. This is easily done by means of a comparison of the "down subscript" and the value set at level r. The result of the algorithm is that the row found at level r can remain empty for several recursion steps, thus speeding up the search procedure.

Some examples

A *lower* triangular *matrix* or *minor* and all its column permutations will be ideally developed due to the normal search procedure (compare with fig. 2.4.3,2).

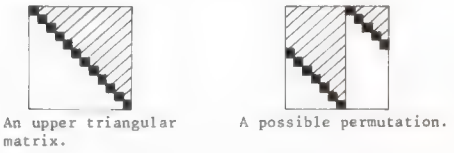


Fig. 2.4.4,3.

An *upper* triangular *matrix* and all its column permutations will be ideally developed due to the fact that all elements of the minor sequence are the last of a column (compare with fig. 2.4.4,1).

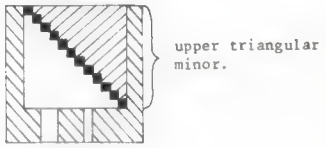


Fig. 2.4.4,4.

If an *upper* triangular *minor* occurs, then a part of the improvement may come from mechanism I, while the rest will come from mechanism II (See fig. 2.4.4,5).

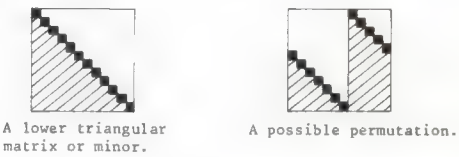


Fig. 2.4.4,5.

III. Static improvements.

If there is only one nonzero matrix element in a row (column) of the skeleton, then the remaining nonzero matrix elements in the associated column (row) can be removed from the skeleton. This should be repeated until no further improvement of the skeleton can be achieved.

2.4.5 The effect of deviations from the statistical model.

The statistical model of 2.1 fits well to the example if a development along the columns is made. The expected number of nonzero matrix elements in each column, ηm, comes close to the actual values, relatively independent of the column chosen. If however a development along the rows is made then the following observations can be made:

*) See fig. 2.3,1 and fig. 2.3,2.

1) The value of η is equal to $\eta_1 = 1.8/20$ for the first 10 rows and equal to $\eta_2 = 4.2/20$ for the last 10 rows.

2) $p\{a(i,j)\neq 0 \mid a(k,j)\neq 0\} = 0$ for the first 10 rows.

i.e. in the first 10 rows there is no more than one nonzero matrix element in a column, and consequently the nonzero matrix elements are arranged in a certain pattern.

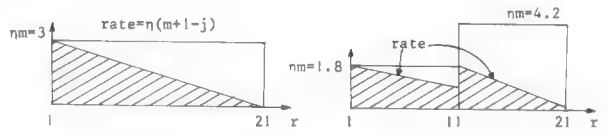
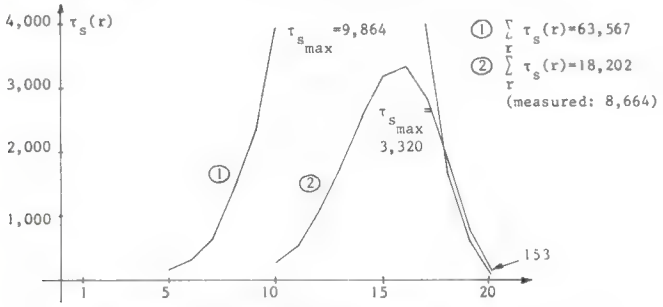


Fig. 2.4.5,1.

A calculation of the $\tau_s(r)$ curve shows that unrealistic low values are obtained, not corresponding with the measurements. Observing the matrix, taking 2) into consideration, clarifies that $\tau_s(10)=2^8=256$. With this value as a starting value it is still possible to give good predictions based on statistical data, as can be seen from fig. 2.4.5,2.



1 Development along the *columns* (see also 2.3.1).

2 Development along the *rows*.

Fig. 2.4.5,2.

2.4.6 Invocation count and time measurements.

The result of the speed up will be given in tabulated form for the example introduced in 2.2. For a more general discussion about invocation count measurements see for example [7].

development along:	*speed up:*	*count:*	*time in seconds IBM 360/50-LCS* *)
column	none	47,218	60
row	none	8,664	10.14
row	I	3,669	4.39
row	I+II	3,392	4.19
row	I+II+III	2,729	3.75

2.4.7 A mixed symbolic-numeric method.

A mixed symbolic-numeric method is implemented in the NETFORM system by C.W. Gits [8]. The idea is to distinguish two parts of the skeleton:

1) A lower part which constitutes of rows with only constant entries.

2) An upper part in which symbolic and constant entries are mixed.

This situation occurs particulary often in NETFORM. The key points of the implementation are described under the assumption that the method is applicable.

1) If a complete minor sequence is found, only the the constant part will be developed.

2) If the recursion reenters the symbolic part of the matrix, then the value of the constant minor computed by 1) is linked to the symbolic term.

3) The determinant is calculated from:

   1: The symbolic minor sequence T'(A).
   2: The value of the constant minor C.

   Note that the value of the sign is imbedded in the constant C. So:

$$t = T'(A)$$

and

$$det = det + t$$

The advantage of the method is that several terms, are contracted in C, which implies that fewer symbolic terms have to be processed by FORMAC. In our example we found 9 symbolic terms with the mixed method, while 119 where found with the previous method. The time required to simplify this result also dropped from ± 4 sec. to neglectable.

2.5 A proposal for a new algorithm.

If $\eta<<1$ then the function $\tau_s(r)$ shows a range of exponential growth and a range of exponential decrease. As a measure for the relative efficiency of the SLE method the ratio

$$R = \frac{\tau_s(m)}{\tau_{s_{max}}}$$

can be taken

$$R = \frac{\tau_s(m)}{\tau_{s_{max}}} = \frac{m!\,\eta^m}{\frac{m!}{(m-(m+1)+\frac{1}{\eta})!\,\eta^{(m+1-\frac{1}{\eta})}}} = \eta^{(\frac{1}{\eta}-1)}\,(\tfrac{1}{\eta}-1)!$$

Note that this ratio is not related to the size of the matrix (see also fig. 2.5,1).

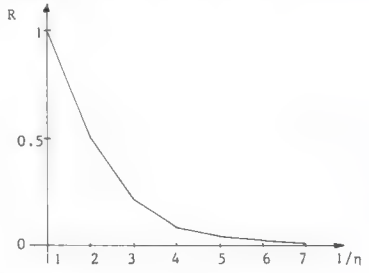


Fig. 2.5,1.

This ratio (R) shows that $\tau_s(m)$ is *necessarily* lower than $\tau_{s_{max}}$ for a given value of $\eta<1$.
This raises the question wheather a development is possible which has a better relative efficiency (R).

All methods discussed so far developed along the *first* row (column) of the minor. If this fixed strategy is replaced by a more flexible, adaptive, strategy which develops along the row (column) with a minimal number of nonzero matrix elements, then a considerable improvement may be expected. For this case the definition of the determinant gives:

*) One count takes ± 0,8 msec. = ± 100 cycles in the LCS.

Development along the i-th row:

$$|A| = \sum_{j=1}^{m} a_{ij}(-1)^{i+j}M_{ij}$$

and

Development across the j-th column:

$$|A| = \sum_{i=1}^{m} a_{ij}(-1)^{i+j}M_{ij}$$

Now two search procedures may be considered:

A: Dynamic optimal row-development.
This method develops along the row with the minimal count.

B: Dynamic optimal row-column development.
This method develops along the row or column with the minimal count. This method has the disadvantage that a mixed symbolic-numeric method is impossible due to the development along the column.

Algorithm A is currently in development.

2.5.1 The effect of the new algorithm expressed in properties of the minor.

The effect of the new algorithm is simpler to describe, since the rules are more general. Cases which would have required several iterations can now be discovered at once. A return from recursion, which does not contribute to the generation of terms, occurs in the case that an empty row (or column) is detected in the minor.

Fig. 2.5.1,1.

Note that the old algorithm will continue to develop the minors until the top row of the minor becomes empty. This implies that in an early stage of the development the reasons for the exponential growth can be removed.

It is interesting to note that an empty row (column) in the minor seldom occurs. This is due to the fact that a row (column) with only one nonzero entry will be taken first in the development.

EXAMPLE:

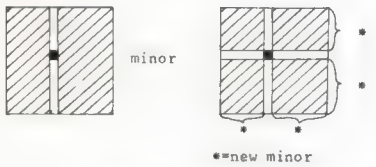


Fig. 2.5.1,2

Only if there are two or more rows (columns) with one nonzero entry, and the nonzero entries are found in the same column (row) then a return from recursion will follow.

EXAMPLE:

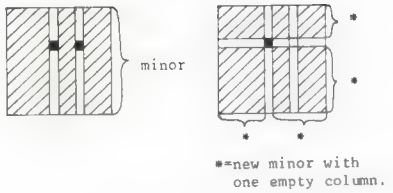


Fig. 2.5.1,3

Note that this situation will rarely occur. If it would not occur at all then it could be proved that the new $\tau_s(r)$ curve would be monotonically increasing.

3. References.

[1] E. Horowitz and S. Sahni: On Computing the Exact Determinant of Matrices with Polynomial Entries. Journal of the ACM, Vol. 22, No. 1, January 1975, pp. 38-50.

[2] E.H. Bareiss: Sylvester's identity and multistep integerpreserving Gaussian elimination. Math. Compt. 22, 103, July 1968, pp. 565-578.

[3] M.L. Griss: The algebraic solution of sparse linear system via minor expansion. Utah computational physics group report no. 32, October 1974 (Revised March 1975).

[4] J. Smit: Introduction to NETFORM. ACM-SIGSAM Bulletin, Vol. 8, No. 2, May 1974, pp. 31-36.

[5] J.D. Lipson: Symbolic Methods for the Solution of Linear Equations with applications to Flowgraphs. Proceedings of the 1968 Summer Institute of Symbolic Computation, IBM Programming Laboratory Report, No. FSC 69-0312, 1969.

[6] A. Antoniou: Realisation of gyrators using operational amplifiers, and their use in RC-active-network synthesis. Proceedings of the IEE, Vol. 116, No. 11, November 1969.

[7] K. Bahr and J. Smit: Tuning an Algebraic Manipulation System through Measurements. ACM-SIGSAM Bulletin, Vol. 8, No. 3, August 1974, pp. 17-23.

[8] C.W. Gits: The development of symbolic matrices using a mixed symbolic-numeric method. BSC report at T.H.T./E.L., P.O.Box 217, Enschede, The Netherlands (February 1975).

Automatic Symbolic Solution of Differential Equations of First Order and First Degree

Peter Schmidt
Institut für Informatik, Universität Bonn
Bonn, Federal Republic of Germany

Abstract

The problem of solving first order, first degree differential equations symbolically is characterized as a heuristic search process. An investigation into the problem of automatically solving such differential equations has resulted in a heuristic program, called EULE. The selection and the realization of the methods for EULE are based on a detailed analysis of the problem domain: the standard work of Kamke (1961), which is representative of the state of knowledge of differential equations, was examined in three different respects: the collected methods of solution, the methods utilized for the collection of differential equations and the structure of these differential equations. The realization of the methods is based on this result and on defined principles which ensure the effectiveness of the program. The effectiveness of EULE can be characterized by the fact that EULE achieved a 'rate of solution' of 90% for Kamke's representative collection of first order, first degree differential equations and a rate of 95% for Murphy's (1960) representative collection. For two collections for training students EULE achieved a rate of 100%.

1. Introduction

At the moment there does not exist in the theory of differential equations[1] a general method for finding symbolic solutions of first order, first degree DEs, i.e. equations like

(1.1) $f(x,y)y' + g(x,y) = 0$.

The most common methods of solution are based on transforming DEs either by substitutions of variables or by multiplication by an expression into a separable, linear or exact DE. They are, respectively, of the form $y'=f(x)g(y)$, $y'=f(x)y+g(x)$ or $f(x,y)y'+g(x,y)=0$, where $f_x=g_y$. These forms immediately reduce to integration problems.

So how should a given DE be treated? Murphy (1960) for example suggests in his treatise the following way: "Go to Part I. It contains most of the methods which have been developed for solving ordinary DEs. If directions are followed, it should be possible to classify the given equation as a special case of one or more general types. When the type has been identified, details of the method for solving it will also be found."

Following Murphy's suggestion and having found a solution one can classify the way of finding the solution as follows:

1) the solution could be found at once without much effort,
2) the solution could be found quite quickly by using characteristics of the DE,
3) the solution could be found by chance and often after much effort.

This classification is based on the following stages in finding a solution:

in the first stage the most common and easily recognizable patterns and sufficient conditions are tested. Here, only a little experience with DEs will suffice.

In the second stage the methods of solution, which have become obvious by examining the DE, are tested. Here, intuition and experience are necessary.

In the third stage a lot of nearly equivalent paths are tested without one's being able to set up a general order among them by using mathematical theory or intuition. The success depends on the order which has been chosen at random and on the patience

1) in the following abbreviated: DEs

involved. As only an extremely small number of DEs can be solved in terms of integrals and elementary functions, the hope of finding a solution decreases more and more. If one does not give up this might be the reason for mistakes which prevent one finding a solution.

Now, the solving of DEs is characterized as a heuristic search. An approach to the problem by a computer program could be suitable.

The following text reports on investigations concerning the problem of automatic symbolic solving of first order, first degree DEs and a program, called EULE, which has been developed for solving such DEs.

This research was made mainly for two reasons: as already pointed out the author regards this domain as an extremely suitable subject for heuristic programming. On the other hand, the already existing program SOLDIER, which Moses (1967) has written in addition to his integrator SIN does not appear powerful enough.

Compared with the program SOLDIER, EULE has certainly the advantage that more methods of solution have been implemented and that the methods have been realized in a more powerful way.

The selection and realization of the methods for EULE are based on a detailed analysis of the problem domain: the standard work of Kamke (1961), which is representative of the state of the knowledge of DEs, was examined in three different respects: the collected methods of solutions, the methods utilized for the collection of DEs (Kamke has found 367 DEs to be the most important), and the structure of these DEs. The realization of the methods is based on this result and on defined principles which ensure the effectiveness of the program. The effectiveness of EULE can be measured in test results which EULE has obtained for several collections of DEs, among which two are representative.

In these respects EULE differs fundamentally from former heuristic programs, such as the programs SAINT (Slagle (1963)) and SIN (Moses (1967), Moses (1971a)) for indefinite integration, the program SOLDIER for solving first order, first degree DEs and the program WANDERER (Wang (1971)) for definite integration. These programs, like EULE, can solve problems in complex sections of analysis by means of formula manipulation, but the basic conception of EULE is the analysis of the total domain (here especially the domain of first order, first degree DEs) and the effectiveness of EULE is found by tests with data representative of the domain.

The following text gives a description of the conception and the results of the analyses (chapter 2), a summary of the program EULE (chapter 3), a selection of test results (chapter 4) and some reflections on the program EULE (chapter 5).

2. Concepts

2.1 Specification of the Problem

In the following text the problem is specified for which EULE has been developed.

EULE shall find general solutions for first order, first degree DEs of the form

(2.1) $f(x,y)y' + g(x,y) = 0$.

An equation $L(x,y;C) = 0$ depending on a parameter C is called a 'general solution' of (2.1), if particular solutions result from (2.1) by substituting numerical values for C and calculating y from the equation. And a function $y=l(x)$ is called 'particular solution' of (2.1), if l is differentiable and the identity $f(x,l(x))l'(x)+g(x,l(x))=0$ holds.

Since first order, first degree DEs and general solutions will be in the foreground of the discussion, the shorter notations DEs and solutions will be used for those terms.

The methods of solution outlined in the introduction are admitted for EULE. These are recognizing a DE as being of one of the elementary forms separable, linear or exact; transforming a DE into such a type by substituting variables, or multiplication with an Euler multiplier. Only those substitutions are regarded which transform the DEs again into a DE of first order and first degree. Therefore EULE is only able to solve those DEs for which a solution exists which can be expressed in form of integrals and the elementary functions. For example this is not possible for the Special Riccati DE

$$y' + ay^2 = bx^\alpha \quad .$$

According to the usual convention a DE is regarded as solved, if the DE problem is reduced to an integration problem. If EULE detects a way of solution, it only gives this way without giving a further solution.

EULE is able to treat DEs as they usually appear. The DEs may contain the following elements:

- an independent and a dependent variable,
- constants (rational numbers, literal constants, the real numbers π and e),
- elementary functions (sin cos tan cot sec csc exp log abs arcsin arccos arctan arccot sinh cosh tanh coth),
- general functions of one variable,
- derivatives of the dependent variable and the general functions,
- the usual arithmetic operations.

2.2 Analyses and Principles for the Selection and Realization of the Methods of Solution

The author has used the standard work of Kamke (1961), which is representative of the state of knowledge of DEs, for his

analysis of the problem domain.

The collection of the 367 Kamke DEs (1.1-1.367) was examined with regard to the question which mathematical methods of solution should be selected. In table 2.1 there can be found detailed information about which methods were used and how often those methods appeared; the numbers in brackets refer to the number of cases in which the method is successful only after a substitution of variables.

	method	frequency
I	separable	10% (10%)
	homogeneous	8% (1%
	linear coefficients	4%
	linear	8% (6%)
	Bernoulli+)	5% (4%)
	exact	5% (1%)
II	multiplier	7%
	type Riccati and particular solution	4%
	substitution to obtain a DE of second order	5%
	reference to another DE in the collection	4% (5%)
	no method, only solution	4%
	reference to the literature	3% (2%)
	different methods	4%

table 2.1: frequency of methods used for the Kamke DEs 1.1-1.367

The data of table 2.1 gives a first rough hint to which methods of solution should necessarily be represented in the program: i) the methods of the group I , ii) the methods of the group II , and iii) methods concerning substitutions of variables.

In the following text the underlying principles are discussed which are mainly responsible for the development of the program EULE. The above-mentioned grouping is utilized for the discussion.

i) In the group I of the table 2.1 those methods of solutions can be found which are considered as the most basic in the literature.

The DEs in Kamke and also in other collections which can be treated by those methods can easily be solved by a person with only a little effort.

To make sure that EULE is able to solve not only the special examples of DEs like those in Kamke but also those of other collections and those resulting from substitutions of variables, the following demand (D1) for the realization of these methods seemed suitable:

demand (D1):

the realization of the methods recognizing the types of group I should be so powerful that within the scope of the structure of the DEs nearly all DEs of those types could be solved (for the realization it is not so important whether the examples of the Kamke collection are complicated or not).

Some problems involved with fulfilling this demand shall be discussed in connection with the problem of recognizing DEs as exact.

A differential equation given in the form (2.1) is recognized as exact, if the expression $f_x - g_y$ has been found to be identical to zero.

Richardson (1968) proved that for sufficiently rich classes of expressions the question of zero-equivalence is undecidable. Such a class of expressions is for example generated by the rational numbers and the two real numbers π and $\log(2)$, the variable x , the operations of addition, multiplication and composition, and the functions sin, cos and abs.

As the functions f and g of the DEs of the form (2.1) may contain those elements, it is not possible to completely recognize the class of the exact DEs. To get still satisfying results concerning the demand (D1) two approaches can be made.

One approach is to confine oneself to such classes for which a zero-equivalence algorithm exists. Moses (1971b) gives a survey of such algorithms. Brown (1969), for example, describes a simplification algorithm for the class of expressions which is generated by the rational numbers, i and π , the variables $x_1, x_2, \ldots, x_n$, the operations of addition, subtraction, multiplication and division, the function exp and the composition of functions. One algorithm for the specific class of radical expressions is described by Fateman (1972).

The other approach is to choose a procedure which can decide zero-equivalence in 'simple' cases, and then to enrich the procedure with specific facilities.

This last approach shows the following advantages: it is easier to realize, extensions are easier to make, and moreover, a successful realization renders possible an insight into the structure of exact DEs which are found in the literature. This approach has the disadvantage , that recognition is not given in all cases.

In order to decide whether the second approach, preserving the demand (D1) , is possible, an analysis of the structure of the DEs in the Kamke collection was made.

+) in many cases the type almost linear was mentioned for Bernoulli DEs

For the 367 Kamke DEs (in expanded notation) the following characteristic features have been found:

- on an average the DEs consist of 4 sum terms (of first level) and less than 4% of the DEs have more than 6 terms,

- those 4 terms in all have on an average 9 factors (of first level), and only about 6% have more than 14 factors,

- about 70% of the DEs do not contain any function;
if they do contain functions, the arguments of those functions either match one of the patterns $ax^b \pm cy^d$ or $x^a y^b$, or they consist of a sequence of functions with x or y as innermost argument (a,b,c,d are literal constants or rational numbers);
in more than 50% of the cases the arguments are only either x or y ,

- about 90% of the DEs do not contain powers of a sum or sums in a denominator;
if such expressions appear, they are in 80% of the cases rational powers of polynomials in x or y ; in about 70% of these cases the polynomials depend only on x or only on y ,

- nearly 90% of the DEs contain as factors (of first level) of the sum terms (of first level) only expressions like

c or x^c or y^c or $f^c(x)$ or $f^c(y)$

or at the most one expression, which is not of this kind;
in more than 70% of those DEs, which contain more than one expression of another kind, only two of these expressions appear and have the following forms

either:	$f(A)$	and	$g(A)$
or:	P^r	and	Q^s
or:	x^A	and	x^B

(c is a literal constant or a rational number; f and g are function designators A and B are any expressions; P and Q are polynomials in x or y; r and s are rational numbers).

Those DEs, which are obtained by substitution of variables, are simple in a similar sense.

Thus the DEs have such a simple structure that the above-mentioned second approach to recognize exact DEs can obviously be preferred.

In order to fulfil the demand (D1) for the realization of the methods of the group I the following way has been chosen: procedures have been developed that should be adequate in the first place to the forms, which have to be recognized, and to the general structure of the DEs, and only in the second place to the special examples of the literature .

ii) The methods of solution which are mentioned in group II of table 2.1 include many possible algorithms and heuristics that can be utilized. The particular solutions of Riccati DEs or the Euler multipliers, which are noted for the Kamke DEs as ways of solution, can generally be computed in different ways. For a person it is not a trivial problem to find such a solution, since there is a variety of possibilities without a criterion for stopping.

For this reason, a suitable realization of those methods can enable a program to find additional ways of solution of that kind for certain DEs. In consequence, DEs might be solved for which methods are noted in literature that are not implemented in the program.

The procedures for the methods of the group II have been designed according to the following demand (D2):

demand (D2):

the procedures for the methods of the group II should be selected and realized in such a way that as few and as general algorithms or heuristics as possible could solve all or at least nearly all the DEs that Kamke solved by one of the methods of the group II.

That demand should render the chosen algorithms or heuristics so powerful that their scope would exceed the special examples of the Kamke collection solved by those methods.

In order to fulfil the demand (D2) the following way has been chosen: at first, the DEs and their particular solutions as well as the Euler multipliers noted in the Kamke collection were listed. Then, using methods mentioned in the literature or own ideas, procedures were looked for which could cover all the listed examples.

iii) The methods concerning substitutions of variables were selected with regard to the aim of increasing the number of DEs which could be solved by the program without implementing too specific procedures. To this end the following demand (D3) seemed suitable:

demand (D3):

the methods concerning substitutions of variables should be selected and realized in such a way that with their aid 'many' Kamke DEs could be solved and especially 'some' Kamke DEs for which methods of the groups I and II did not succeed.

Like demand (D2), this demand should make it possible that for certain DEs new ways of solution could be found.

In order to choose methods for EULE the following examinations (E1)-(E4) of the Kamke DEs were performed:

(E1) do the methods for Riccati or Abel DEs, noted in the text part of Kamke's treatise, apply to solve the corresponding DEs in the collection?

(E2) Does the collection contain patterns of DEs, which subsume other DEs of the collection?

(E3) Do certain substitutions exist, which provide a solution in a large number of cases?

(E4) Can heuristics be found that connect 'characteristic' terms in DEs with successful substitutions?

3. The Program EULE

The program EULE is not based on any existing algebraic manipulation system. EULE, together with all its components, has been developed only for the purpose of solving ordinary DEs. The main decisions concerning the representation of expressions and the simplification are based on this goal.

EULE is able to treat any complex expressions formed by the elements mentioned in section 2.1 . A central simplifier (see section 3.1) can transform the expressions into different forms. Methods of solution start from one of those forms. Apart from that general representation of expressions there are also more specific representations possible. They are used for the design of more efficient algorithms, which apply for DEs lying in more specific classes of expressions. Such and also further more general facilities of EULE are (partly implicitly) described in connection with the description of the methods of solution employed.

EULE is programmed in PL/I and consists of about 8500 statements.

EULE can read DEs in a form which corresponds to the usual mathematical notation. The most important restrictions are: i) all terms not identical to zero have to be placed to the left side of the equation, and ii) special names have to be used:

X for the independent variable,
Y for the dependent variable,
A0,A1,A2,... for general constants,
F0,F1,F2,... for general functions,
PI and E for π and e .

The following Kamke DEs

$y' + ay = ce^{bx}$ (Kamke 1.2) ,

$y' + f'(x)y = f(x)f'(x)$ (Kamke 1.10),

$x^4(y'+y^2) + a = 0$ (Kamke 1.181)

are for example transformed into the notation:

```
Y' + A0*Y - A2*EXP(A1*X) = 0 ,
Y' + F0'(X)*Y - F0(X)*F0'(X) = 0 ,
X**4*(Y'+Y**2) + A0 = 0 .
```

The following text contains a brief description of the central simplifier (section 3.1), a brief description of the methods of solution implemented in EULE (section 3.2) and the description of the strategy employed for solving DEs (s. 3.3).

3.1 The Central Simplifier of EULE

According to the classification of Moses (1971b) the central simplifier of EULE is a 'liberal' one. Because of its importance for the realization of the methods of solution a partially detailed description is presented.

The following forms (F1) - (F5) of a DE may be of importance for the process of solving a DE. They can be shortly characterized by:

(F1) performing all automatic transformations;

(F2) in addition to (F1) applying the distributive law concerning + and * ;

(F3) in addition to (F2) expanding exponentiations of sums for positiv integer exponents up to a given maximum value;

(F4) in addition to (F1) factoring of exponentiations of sums depending on x and y into a product of two factors, of which one depends only on x and the other only on y (the factoring is performed by the routine employed for the recognition of separable DEs);

(F5) in addition to (F1) reducing the expression to a common denominator and removing the denominator.

Automatic transformations are (they are recursively performed):

1) transformations concerning operators
($+a \rightarrow a$; $a-b \rightarrow a+(-b)$; $a/b \rightarrow a*b^{-1}$;
$-(-a) \rightarrow a$; $(-a)*b=a*(-b) \rightarrow -a*b$;
$(a\overset{*}{+}b)\overset{*}{+}c \rightarrow a\overset{*}{+}(b\overset{*}{+}c)$; $(-a)^{n/m} \rightarrow -a^{n/m}$, if n and m are odd integers; $(-a)^{n/m} \rightarrow a^{n/m}$, if n is an even integer) ;

2) the usual transformations concerning 0 and 1 in sums, products and exponentiations;

3) transformations concerning expanding
($-(a+b) \rightarrow -a+(-b)$; $(a*b)^c \rightarrow a^c*b^c$;
$a^{b^c} \rightarrow a^{b*c}$) ;

4) ordering of terms in sums and products according to the following rules:

i) terms in sums and products decrease in in complexity in a left-to-right scan,

ii) for the atomic expressions Y',Y,X,A_i and the rational numbers the complexity decreases from left to right;

iii) the order of complexity for any expressions is recursively defined; the corresponding table is omitted here; instead of this table some examples are presented: the complexity again decreases from left to right:
Y', Y^2,SIN(Y),Y+X,SIN(SIN(X))+X+3,X,$A0^2$.

This sorting accomplishes two things a) terms which can be collected according to 5) are placed side by side and

b) terms containing Y' precede those without any Y' ; terms containing Y precede those without Y' and Y ; terms containing X precede constant terms.

5) Collecting terms

(in sums, terms only differing from a rational factor are collected; in products, factors with identical bases are collected; purely rational terms are calculated (the numbers are always small so that there does not arise a problem)).

6) Transformations concerning functions

(exp(log(a))=log(exp(a)) → a; analogous for sin and arcsin,...;

sin(-a[+b]) → -sin(a[+(-b)]); analogous for the other trigonometric functions and the hyperbolic functions).

There are some more transformations possible concerning functions. These are not performed automatically, but they can be controlled in each simplification step. In the current strategy of EULE the following transformations are generally performed (and only few modifications arise):

A) $\log(a^b[*c]) \rightarrow b*\log(a[*c^{1/b}])$;

$\exp(a)*\exp(b) \rightarrow \exp(a+b)$;

$\exp(-a[+b]) \rightarrow \exp^{-1}(a[+(-b)])$;

B) $\sin(2a) \rightarrow 2\sin(a)\cos(a)$; analogous for the other trigonometric functions ;

C) $\cot(a) \rightarrow \tan^{-1}(a)$;

D) $\log(a)+\log(b) \rightarrow \log(a*b)$;

$\sin^2(a)+\cos^2(a) \rightarrow 1$; $1-\sin^2(a) \rightarrow \cos^2(a)$; $1-\cos^2(a) \rightarrow \sin^2(a)$;

these transformations are performed by a procedure which recognizes the terms to be collected even if like factors arise;

E) if the terms $\sin^2(a)$ and $\cos^2(a)$ appear with only rational factors, then $\cos^2(a)$ is replaced by $1-\sin^2(a)$.

3.2 Methods of Solution

The methods of solution employed by EULE are described in this section. Only the main features can be discussed; for more details see the complete report (Schmidt (1976)). The description of the methods is grouped as in section 2.2 .

For the description of the procedures it is presumed that the parts f and g of a DE in the form

(3.1) $f(x,y)y' + g(x,y) = 0$

are given in any of the forms (F1)-(F5).

3.2.1 Recognizing the Elementary Forms

A person usually solves DEs of these types (separable, homogeneous, liner coefficients, linear, Bernoulli, almost linear, exact) at a glance or by a little calculation. According to the concepts (see section 2.2) these methods are realized in EULE in such a way that, generally, they fail only then if the person does not also recognize the type readily.

Separable DEs are of the form

$f_0(y)g_0(x)y' + f_1(y)g_1(x) = 0$.

In general, a routine is needed which is able to factor expressions into a product of which one factor depends only on x and the other only on y .

EULE employs a factorization procedure which greatly uses the internal sorting order. The procedure is only able to factor expressions that do not have any sum term which depends both on x and y. For such an expression A the representation

(3.2) $A=Y_1X_1C_1+Y_2X_2C_2+\ldots,Y_LX_LC_L$

can easily be obtained, where Y_l (X_l) consists of the factors of the lth term of A which depend of y (x), and C_l consists of the constant factors (for $l=1,2,\ldots,L$).

Because of the sorting order (here ">" denotes "more complex"; "=" denotes "equal" complexity, which implies identical representation) the following inequalities hold for $l=1,\ldots,L-1$:

(3.3) $Y_l \geq Y_{l+1}$;
if $Y_l=Y_{l+1}$ then $X_l \geq X_{l+1}$;
if $Y_l=Y_{l+1}$ and $X_l=X_{l+1}$ then $C_l \geq C_{l+1}$.

Let N denote the number of the Y_l which are identical to Y_1 , and let M denote the integer part of the quotient L/N . Then the following conditions are sufficient for the existence of a factorization of (3.2) :

(3.4)
M is an integer ;
for $m=1,\ldots,M$: $Y_{(m-1)N+1}=Y_{(m-1)N+n}$ with $n=1,\ldots,N$;
for $n=1,\ldots,N$: $X_n=X_{(m-1)N+n}$ with $m=1,\ldots,M$;
for $m=1,\ldots,M$ and for $n=1,\ldots,N$:
$C_{(m-1)N+n}=C_{x,n}*C_{y,m}$, where
$C_{x,n}$ is chosen as C_n and
$C_{y,m}$ is chosen as $C_{(m-1)N+1}/C_1$.

If the conditions (3.4) hold, then the factorization

$$A = \left(\sum_{n=1}^{N} C_{x,n}X_n\right)*\left(\sum_{m=1}^{M} C_{y,m}Y_{(m-1)N+1}\right)$$

results.

This procedure involves testing for identity and calculating the products $C_{x,n}C_{y,m}$ before they are compared with $C_{(m-1)N+n}$ $(m=1,\ldots,M;n=1,\ldots,N)$.

The comparison is performed with the product being in form (F3).

The success of this procedure is bound up with the simplifier. It is easy to see that this procedure can successfully factor polynomials in x and y with rational coefficients. An example of a more complicated expression which the procedure can also factor is the expression

$(2/\sin(y)+y^2/a+(y+4)^{-2})(x^2\cos(x)+3/x+4a^2)$

in an expanded notation.

The procedure fails for example for the two expressions

$yx+y(a+1)^{1/2}+x(a+1)^{1/2}+\underline{a}+\underline{1}$ and

$\underline{yxa^2}+\underline{yxa2}+\underline{yx}+ya+y+xa+x+1$.

Their terms are ordered according to the internal order. The cause of the failure is that terms (underlined) appear which differ only with respect to a constant factor. The sum of those factors is not factorized any more.

As only very few DEs of the Kamke collection (not only of the separable DEs) have the above-mentioned attribute, those cases were not additionally treated.

EULE recognizes a DE of the form (3.1) as separable, if f and g can be factored. And an expression can be factored by EULE, either if the procedure described above yields a factorization, or if the expression is an exponentiation and the base can be factored. In addition, if sums appear as arguments of the function exp, the functional equation of exp is applied.

For nearly all separable DEs of Kamke, factorization is not necessary at all. DEs of Kamke which EULE recognized as separable are for example

$y'=((y^2+1)(x^2+1))^{1/2}$ (Kamke 1.60) ,

$y'-\exp(x-y)+\exp(x)=0$ (Kamke 1.75) .

In order not to destroy patterns the test for separable (as well as for homogeneous and linear coefficients) is performed with f and g being transformed successively according to (F1), (F2) and (F3).

DEs of the form (3.1) are <u>homogeneous</u>, if f and g are homogeneous functions of the same degree. By making the substitution $\bar{y}=y/x$ one obtains a separable DE.

In order to recognize homogeneous DEs EULE employs a recursive procedure, which yields the degree of homogeneity for any expression. In order to support the recognition of homogeneity, terms involving the log function are collected according to the functional equation (see section 3.1).

Examples recognized by EULE are:

$xy'+a(y^2+x^2)^{1/2}-y=0$ (Kamke 1.113) ,

$xy'\cosh(y/x)+2x*\sinh(y/x)-y*\cosh(y/x)=0$ (Kamke 1.349) .

DEs of the type <u>linear coefficients</u> are of the form

$$y' = k\left(\frac{ax+by+c}{Ax+By+C}\right) .$$

DEs of that type can easily be transformed into homogeneous or separable DEs.

EULE is able to recognize the usual forms of this type which appear in the literature. Expanded expressions can be recognized as squares or cubes of linear expressions.

Examples (the last three are not of Kamke) recognized by EULE are

$(ay+bx+c)^2y'+(Ay+Bx+C)^2=0$ (Kamke 1.292),

$y'+\sin(\cos((ax-y+2)^2))+b=0$,

$y'-(y+x)^a*(y-x)^{-a}=0$,

$(3x+4y+b+1)^2y'+y^2+x^2-2xy=0$.

But EULE does not recognize the DE

$y'-3(1-x+y)-(x-y)^2=0$

as a DE of the type linear coefficients, because there are two terms each of which is linear in x and y .

<u>Linear</u> DEs are of the form

$f_0(x)y'+f_1(x)y+f_2(x)=0.$

<u>Bernoulli</u> DEs are of the form

$f_0(x)y'+f_1(x)y+f_2(x)y^C=0$ ($C\neq1$, constant);

by making the substitution $\bar{y}=y^{C-1}$ one obtains a linear DE.

<u>Riccati</u> DEs are of the form

(3.5) $f_0(x)y'+f_1(x)y^2+f_2(x)y+f_3(x)=0;$

if $1(x)$ is a particular solution and if the substitution $y=1(x)+1/\bar{y}$ is made, there results a linear DE.

EULE tests these types in the form (F3). The realization is obvious. Bernoulli DEs are also recognized, if a power of y appears as factor in the part with y' (see example below).

Examples (linear, Bernoulli, Riccati) recognized by EULE are:

$(x^2-1)y'+2xy-\cos(x)=0$ (Kamke 1.154) ,

$2x^2yy'+y^2=2x^3+x^2$ (Kamke 1.258),

$(x^2-1)y'+a(y^2-2xy+1)=0$ (Kamke 1.157).

<u>Almost linear</u> DEs are of the form

$f_0(x)h'(y)y'=f_1(x)h(y)+f_2(x)$.

By the substitution $\bar{y}=h(y)$ one obtains a linear DE.

EULE tests this type in the form (F3). EULE employs the factorization routine described above.

Examples recognized by EULE are:

... (Kamke 1.258) (see above) ,

$(x+a)(\sin(y)-3y^2)y'=(\sin(x)+1)*$
$(2\cos(y)+2y^3)+4x+1$.

A DE of the form (3.1) is exact, if the identity $f_x-g_y=0$ holds.

In order to recognize exact DEs, EULE calculates the derivatives f_x and g_y and transforms the difference of the expressions according to form (F3). If this yields 0 the type is recognized.

Examples recognized by EULE are:

$(y^2+x^2+a)y'+2xy=0$ (Kamke 1.273),

$(x*\cosh(y)+\sinh(x))y'+y*\cosh(x)+\sinh(y)=0$ (Kamke 1.348)

3.2.2 Calculating Particular Solutions of Riccati DEs and Euler Multipliers

Both methods have a wide margin for realization. The selection is discussed in the complete report.

EULE may calculate particular solutions of Riccati DEs in two different ways.

A new algorithm has been developed which applies to DEs of the form (3.5), if the f_i (i=0,1,2,3) are of the form

(3.6) $r_1x^{s1}+\ldots+r_nx^{sn}$ (r_j,s_j rational).

The algorithm can then compute the particular solutions of the form (3.6). A correct description is lenghty and therefore omitted.

This algorithm is based on the following considerations: if one replaces y in the DE by a solution 1(x), the terms with identical powers add up to zero; also those with the largest power. Since those terms are generated by the highest term (i.e. the term with the largest power) of 1(x), equations can be set up, from which either the possible highest terms of 1(x) can be computed or at least the possible powers. Continuing, one obtains a set of DEs, one of which has the expression 1(x)-highest term of 1(x) as a particular solution.

The algorithm also applies to more general Riccati DEs, except if literal constants appear as powers of x.

For example, EULE calculated the particular solutions

$\pm ax^{1/2}-x/2$ of the DE

$2x^2y'-2y^2-3xy+2a^2x=0$ (Kamke 1.164).

If this algorithm fails, EULE employs a heuristic procedure to obtain a particular solution. Its underlying idea is that by replacing y in (3.5) by a particular solution 1(x) some terms are generated which add up to the negative value of, for example, the first sum term of f_3.

If $f_i=f_{i1}+\ldots+f_{in_i}$ (i=0,1,2,3), then EULE will successively test the solutions:

(a) $1(x)=K*f_{31}/f_{2j}$ $(j=1,\ldots,n_2)$,

(b) $1(x)=K*(f_{31}/f_{1j})^{1/2}$ $(j=1,\ldots,n_1)$,

(c) $1(x)=K*\int f_{31}/f_{0j}dx$ $(j=1,\ldots,n_0)$,

where K is an additional parameter.

Having replaced y by 1(x) the resulting expression is transformed according to (F3). Using the attribute of EULE's sorting order that terms differing only from a constant factor are placed side by side, one can easily obtain a (quadratic, linear or improper) equation for K from the highest terms of the expression. Now, K is calculated and K is replaced by its value(s). The resulting expression is transformed according to (F3), and if it yields zero, a particular solution is found.

The integration in case (c) can only be performed, if the integrand is one of the following expressions:

x^C (C is a constant expression), $\sin(x)$, $\cos(x)$, $\sin^{-2}(x)$, $\cos^{-2}(x)$, $\exp(x)$ or a derivative of one of the general functions. Before integrating, constant factors are removed.

If the integrands for any j in the case (c) contain two or more factors depending on x, additional solutions are tested: successively, one of the factors forms the integrand and the others are set before the integral. The underlying idea is: if the solution is a product, of which at least two factors depend on x the differentiation yields at least two terms. It might be that only one of these terms contributes to make f_{31} zero ((Kamke 1.33) is an example, see below).

Using this heuristic EULE calculated the following particular solutions:

$y'-y^2-y*\sin(2x)-\cos(2x)=0$ (Kamke 1.22)

particular solution: $\sin(x)/\cos(x)$;

$y'=f'(x)/g(x)y^2-g'(x)/f(x)$ (Kamke 1.33)

particular solution: $-g(x)/f(x)$

EULE can calculate Euler multipliers in two different ways.

First, EULE employs a heuristic procedure which applies to any DE of the form (3.1). EULE attempts to find multipliers, which are functions of

x or y or x^iy^j or $x^i\pm y^j$ (i,j integer)

A DE of the form (3.1) has an Euler multiplier, which is a function of a given expression w=w(x,y), if the expression

(3.7) $$\frac{g_y-f_x}{fw_x-gw_y}$$

is a function of w. In that case (3.1) has the Euler multiplier

$M=\exp(\int W(w)dw)$,

where W denotes the expression (3.7).

The expressions above noted for w are suitable for the Kamke DEs.

Realizing this multiplier method involves two problems. The major problem is to simplify the fraction (3.7), and the minor problem is to recognize the simplified expression as being a function of such a w.

Examining the Kamke DEs with respect to the first problem, a simplifying procedure seems to be necessary which also applies to expressions involving literal constants and functions. A simple heuristic has been found which turned out to be very effective. The effectiveness follows from two facts. Firstly, the heuristic proved to be sufficient for the DEs of Kamke as well as for the DEs of other collections which were solved in the literature by a corresponding multiplier. Secondly, many additional multipliers could be found by EULE.

The heuristic for simplifying the fraction (3.7) is as follows: the numerator and the denominator are separately transformed according to the form (F3). Suppose that in both expressions all factors (of first level) of the sum terms (of first level) like x^C or y^C have a positive integer exponent C. In that case, the numerator and the denominator can be represented as

$P_1*F_1+\ldots+P_n*F_n$ and $Q_1*G_1+\ldots+Q_m*G_m$,

where P_i and Q_j are polynomials in x and y with rational coefficients, and where F_i and G_j consist of the other factors of the corresponding terms. After having removed common factors separately of the $P_1,\ldots,P_n$ and the $Q_1,\ldots,Q_m$, a division algorithm analogous to that for polynomials is applied to the two expressions (the expression having more terms is taken to be the dividend). This division procedure may fail, if two terms P_i*F_i and P_j*F_j are identical in P_i and P_j. That case appeared as being not important.

The minor problem in realizing this multiplier method is recognizing an expression as being a function of x^iy^j or x^i+y^j or x^i-y^j (i,j integer). For this purpose EULE employs a recursive procedure. Instead of a description some typical examples are presented: the expression $((-y)+x+a+1)^{1/2}*(y*3+(-x*3))^{-1}$ is recognized as a function of x-y; the expression $(-y*a)+y+x*a+(-x)$ is not recognized as a function of x-y, because no other ordering of the sum terms is performed than that of the simplifier. The expression $y^{8/3}x^{4/3}+y^2xa+3$ is recognized as being a function of y^2x.

Examples:

$$\frac{(2ay^3+3axy^2-bx^3+cx^2)y'-ay^3+cy^2+3bx^2y}{+2bx^3=0}$$ (Kamke 1.313)

EULE calculated the multiplier $(x+y)^{-2}$. The fraction corresponding to (3.7)

$$\frac{6ay^2-2cy-6bx^2+2cx}{-3ay^3-3axy^2+cy^2+3bx^2y+3bx^3-cx^2}$$

had to be simplified to $-2/(x+y)$, before the above fraction could be recognized as a function of x+y.

(Kamke gives the solution and a reference to the literature.)

$$(xy^4+2x^2y^3+2y+x)y'+y^5+y=0$$ (Kamke 1.327)

Like Kamke, EULE calculates the multiplier $(xy^3-1)^{-2}$. The fraction corresponding to to (3.7)

$$\frac{-4y^4+4xy^3}{2xy^7-2x^2y^6-2y^4+2xy^3}$$

had to be simplified to $-2/(xy^3-1)$, before the above fraction could be recognized as a function of xy^3.

(EULE calculated also another way of solution. By interchanging the variables x and y one obtains a Riccati DE which has the particular solutions y^{-3} and $-y$.)

$$3xy'-3x*\log(x)y^4-y=0$$ (Kamke 1.132)

EULE calculated the multiplier y^{-4} for this Bernoulli DE. The fraction

$$\frac{12xy^3\log(x)+4}{-3xy^4\log(x)-y}$$

had to be simplified to $-4/y$, before the above fraction could be recognized as a function of y.

Besides the methods discussed above, EULE employs an algorithm for calculating multipliers of the form x^my^n (m,n rational numbers) for rational DEs. A DE is called 'rational', if its sum terms are either of the form rx^sy^t or of the form rx^sy^ty', where r,s,t are rational numbers.

A DE of the form (3.1) has the Euler multiplier x^my^n, if the identity $nxg-myf=xy(f_x-g_y)$ holds.

Supposing a rational DE, this identity gives rise to a system of linear equations for m and n, which can easily be solved.

Like Kamke, EULE calculated the multiplier $x^{-27}y^{-16}$ for the DE

$x(y^2-3x)y'+2y^3-5xy=0$ (Kamke 1.293).

3.2.3 Methods Concerning Substitutions of Variables

If EULE recognizes 'characteristic' expressions in a DE, it suggests corresponding substitutions of variables. Two kinds are regarded.

First, if terms like $A(x,y)^C$ or $F(A(x,y))$ appear, EULE suggests the substitution $y=A(x,y), x=x$ (A(x,y) denotes

an expression depending on x and y, C denotes a constant expression and F denotes a function designator).

Secondly, if terms like F(A(y)) appear, EULE suggests the substitution $\bar{y}=F(A(y))$, $\bar{x}=x$ (F and A like above).
If F is for example the function sin, in addition, EULE suggests the substitutions $\bar{y}=\cos(A(y))$, $\bar{y}=\tan(A(y))$ and $\bar{y}=\tan(A(y)/2)$. The other trigonometric functions are treated analogous.

EULE suggested the substitutions $\bar{y}=x^2y$, $\bar{y}=y+x$, $\bar{y}=\tan(y)$ for the three DEs

$xy'-y(x*\log(x^2/y)+2)=0$ (Kamke 1.120) ,

$((y+x)^{1/2}+1)y'+1=0$ (Kamke 1.334) ,

$(x^2+1)y'+x*\sin(y)\cos(y)-x(x^2+1)*\cos^2(y)=0$ (Kamke 1.152),

and reduced them to a separable, separable, linear DE.

In the Kamke collection two types of DEs appear, which often apply to other Kamke DEs. EULE employs procedures to recognize these two patterns:

$y'=ay^n+bx^{n/(1-n)}$ (Kamke 1.52) and

$y'=\frac{f^{1-n}g'}{(ag+b)^n}y^n+\frac{f'}{f}y+fg'$ (Kamke 1.53) ,

where $f=f(x)$ and $g=g(x)$. By a suitable substitution these DEs reduce to separable DEs.

EULE solved the two DEs by these methods:

$x^2(y'+ay^2)=b$ (Kamke 1.143) ,

$y'=a^nf^{1-n}g'y^n+f'/fy+fg'$ (Kamke 1.54).

3.3 The Strategy of EULE

The main aspect for designing the strategy for EULE was to ensure that EULE could solve as many Kamke DEs as possible. But only methods with a large scope should be selected.

Computer tests were performed in order to decide several points in question.

a) Calculating particular solutions of Riccati DEs proved to be a sufficiently quick process. No interruption and resumption after having applied another method is necessary.

b) Only very few multipliers being functions of x^i+y^j or x^iy^j (i,j positive integer) with i or j greater 2 were found in a systematic test by EULE.

c) Several substitutions of variables were applied to a large set of Kamke DEs. It turned out that only the substitutions $\bar{y}=yx,\bar{x}=x$ and $\bar{y}=y/x,\bar{x}=x$ and $\bar{y}=x,\bar{x}=y$ fulfilled the demand (D3) (see section 2.2). The substitutions $\bar{y}=1/y,\bar{x}=x$ and $\bar{y}=y^2,\bar{x}=x$ and $\bar{y}=y^2+x^2,\bar{x}=x$ were rejected.

d) The form (F3) used in connection with the choice of the modifiable transformations with respect to functions (see section 3.1) proved to be suitable for the most applications.

Relying on those facts EULE treats a given DE as follows:

A1) EULE tests the DE successively in the forms (F1),(F2),(F3) to recognize it as being of one of the types: separable, homogeneous or linear coefficients.

A2) EULE tests the DE in the form (F3) to recognize it as being of one of the types: linear, Bernoulli, Riccati, almost linear, Kamke 1.52 or Kamke 1.53.
If the DE is a Riccati DE the algorithm and afterwards the heuristic for calculating particular solutions are applied.

A3) EULE tests the DE in the form (F3) to recognize it as being of the type exact, or to find multipliers which are of the form x^my^n or are functions of x or y.

A4) The variables x and y of the DE are interchanged and the test like A2) is performed.

B) The substitutions of variables are performed which EULE suggests, if characteristic expressions appear in the DE (see section 3.2.3) .
The tests A1)-A4) are performed after each substitution.

C) The substitutions $\bar{y}=yx$ and $\bar{y}=y/x$ are performed. The tests A1)-A4) are performed after each substitution.

D) EULE uses the DE in the form (F3) in order to calculate multipliers which are functions of $x^i\pm y^j$ or x^iy^j (i,j=1,2,3).

4. Test Results

EULE has been tested on four collections of DEs

Kamke (1961)	: 1.1-1.367 ,	*3,39* ;
Murphy (1960)	: 1-751 ,	*1,35* ;
Ince (1956)	: 1-122 ,	*1,0* ;
Spiegel (1958)	: 1-80 ,	*4,0* ;

the first number between the asterisks is the number of 'not attempted' DEs (they did not fit with the conventions of EULE); the second number between the asterisks is the number of 'not elementarily solvable' DEs (they do not have a solution in form of integrals and the elementary functions).

The collections of Kamke and Murphy are representative collections of DEs (i.e. both authors collected those DEs which seemed to be the most important). The collections of Ince and Spiegel are used for training students.

Since EULE has been developed on the basis of the Kamke DEs, the test results on the representative collection of Murphy should give a more objective criterion for the effectiveness of EULE.

Rate of Solution

The effectiveness of EULE can be characterized best by its 'rate of solution'. The rate of solution, that is the percentage of solved DEs of a collection after not attempted DEs and not elementarily solvable DEs have been removed. EULE achieved a rate of solution of 90% for the representative collection of Kamke and a rate of 95% for the representative collection of Murphy (see table 4.1).

collection	number of tested DEs	EULE's rate of solution
Kamke	333	90%
Murphy	715	95%
Ince	121	100%
Spiegel	76	100%

table 4.1: rate of solution

Moses' program SOLDIER achieved for the 76 Spiegel DEs which are easily solved a rate of solution of 89%; EULE achieved the same percentage by employing only its routines for exactness and for calculating multipliers (see table 4.2).

Effectiveness of Single Methods

Kamke (Murphy) solves 171 (505) DEs by recognition of one of the elementary types (separable,...). EULE also recognized those types in 100% (99,4%) of the cases.

Kamke (Murphy) solves 18 (23) Riccati DEs by giving a particular solution. EULE calculated in 100% (100%) of the cases the same solutions.

Kamke (Murphy) solves 25 (71) DEs by noting a multiplier. EULE calculated in 92% (90%) of the cases the same multipliers.

The above-mentioned results, especially those for the Murphy collection, prove the demands (D1) and (D2) (see section 2.2) for realizing the methods of solution to be suitable.

Additional Ways of Solution

For very many DEs EULE has found an additional way of solution, which is not noted in the literature. The report (Schmidt (1976)) contains a list of about 125 Kamke DEs for which EULE found also another way of solution.

Kamke (Murphy) notes for 45% (43%) of the 40 (54) elementarily solvable Riccati DEs a particular solution. EULE found such a way of solution in 83% (76%) of the cases.

EULE solved a large portion of the collections by using only multiplier methods. The results are listed in table 4.2:

collection	number of tested DEs	DEs solved (in %) by exactness + multiplier methods
Kamke	333	16%+40%=56%
Ince	121	9%+71%=80%
Spiegel	76	18%+71%=89%

table 4.2: EULE's rate of solution by using only the procedures for exactness and for calculating multipliers

Some examples are presented for which EULE calculated alternative ways of solution. Some of these ways seem to be easier to find than the ways noted in Kamke.

$x^4(y'+y^2)+a=0$ (Kamke 1.181)

EULE calculated for this Riccati DE the particular solutions $\pm(-a)^{1/2}x^{-2}+x^{-1}$.
Kamke suggests the substitution $u(x)=x^2y$ and subsequently the substitution $v(x)=u-x$ to obtain a separable DE.

$x^3y'-x^6y^2-(2x-3)x^2y+3=0$ (Kamke 1.173)

EULE calculated for this Riccati DE the particular solutions x^{-3} and $-3x^{-3}$.
Kamke suggests the transformation $y=x^{-3}u'u^{-1}$ to obtain a DE of second order.

$xy'\log(x)-y^2\log(x)-(2\log^2(x)+1)y-\log^3(x)=0$ (Kamke 1.194)

EULE calculated for this Riccati DE the particular solution $-\log(x)$.
Kamke suggests the substitution $y=u(s), s=\log(x)$ and subsequently the substitution $v'(s)=-vu$ to obtain a DE of second order.

$y'\sin(x)-y^2\sin^2(x)+(\cos(x)-3\sin(x))y+4=0$ (Kamke 1.195)

EULE calculated for this Riccati DE the particular solutions $-4/\sin(x)$, $1/\sin(x)$.
Kamke suggests the substitution $u(x)=y*\sin(x)$ to obtain a separable DE.

$x(y^2+x^2-a)y'-y(y^2+x^2+a)=0$ (Kamke 1.294)

EULE calculated the multipliers $M_1=(y^2-x^2+a)^{-2}$ and $M_2=(xy)^{-2}$ so that the solution is immediately given without integrating: $M_1/M_2=\text{constant}$.
Kamke suggests the substitution $u(x)=xy$, $v(x)=y/x$ to obtain two separable DEs

$(10x^3y^2+x^2y+2x)y'+5x^2y^3+xy^2-3y=0$ (Kamke 1.304)

EULE found the substitution $\bar{y}=yx$ to obtain a separable DE.
Kamke notes a specific pattern from which a multiplier results.

$(y-x^2)y'+4xy=0$ (Kamke 1.218)

EULE found five different ways of solutions: the substitutions $\bar{y}=y/x$ or $y=\bar{y}^{-2}$ to obtain a homogeneous DE; the interchange of the variables to obtain a Bernoulli DE; the multipliers $(y+x^2)^{-3}$ and $y^{-3/2}$. Kamke notes the substitution $y=x^2u(x)$ to obtain a separable DE.

Computer Times

The tests with EULE were run on an IBM 370/168.

In general, if EULE is able to solve a DE, it rarely needs more than 15 seconds. Moreover, 5 seconds are generally sufficient, if EULE computes a solution in one of the steps A1) - A4) or B) of the strategy (see section 3.3). And half a second is enough for recognizing one of the elementary forms.

Below, some times are presented (only the number of the DE is mentioned and the method of solution - the DE itself is noted in the text, where the method is described). EULE used the strategy described in section 3.3 .

Kamke 1.348	(exact)	0.5 seconds
Kamke 1.164	(Riccati)	2.5 seconds
Kamke 1.33	(Riccati)	3.2 seconds
Kamke 1.313	(multiplier)	27.0 seconds
Kamke 1.293	(multiplier)	1.0 seconds
Kamke 1.120	(substitution)	1.5 seconds

EULE stopped after one minute and 20 seconds for the DE

$(3x+2)(y-2x-1)y'=y^2-xy+7x^2+9x+3=0$ (Kamke 1.247)

without having solved it.

5. Conclusions

The DEs of Kamke , which EULE was not able to solve, can be classified as follows: i) Kamke solves the DE by a substitution , which EULE could not find. An example is the DE Kamke 1.247 (just above) which Kamke solves by the substitution $x=u-2/3, y=v(u)-1/3$. ii) Kamke solves the DE by a sequence of substitutions. For example, Kamke suggests for the DE

$x^7y'+2(x^2+1)y^3+5x^3y^2=0$ (Kamke 1.185)

the three successive substitutions $u(x)=1/y$, $xv(x)=x^3u+1$ and the interchange of the variables x and v to obtain a linear DE. iii) Kamke solves the DE by a methods, which EULE does not employ. For example, a DE is transformed into a DE of second order and solved by a corresponding method.

Persons who publish collections of DEs could utilize a program like EULE to make their work easier. Murphy remarked in the preface to his book:"I have solved each equation in Part II (i.e. the collection) by the methods indicated. This work involved several thousand pages of notes."

Moreover a tool like EULE might help to unify the ways of solution given in the literature. For example, the four DEs of Kamke 1.314,1.210,1.207,1.258 are all of the type $f(x)y^iy'+g(x)y^{i+1}+h(x)=0$, which is an almost linear and a Bernoulli DE. Kamke suggests the following ways of solution: Bernoulli (1.314), almost linear (1.210), Bernoulli and almost linear (1.207), substitution to obtain a separable DE (1.258).

References

Brown, W.S. (1969): "Rational Exponential Expressions and a Conjecture Concerning π and e", Amer. Math. Monthly, vol.76, pp. 28-34, Jan. 1969.

Fateman, R.J. (1972): "Essays in Algebraic Simplification", Report MAC TR-95, Project MAC,MIT, Cambridge, Mass., Apr. 1972.

Ince, E.L. (1956): "Die Integration gewöhnlicher Differentialgleichungen", Bibliogr. Inst., Mannheim, HTB Nr. 67.

Kamke, E. (1961): "Differentialgleichungen-Lösungsmethoden und Lösungen, 1. Gewöhnliche Differentialgleichungen", Akad Verl. Ges. Geest&Portig K.G., Leipzig, 7th edition.

Moses, J. (1967): "Symbolic Integration", Report MAC TR-47, Project MAC, MIT, Cambr., Mass., Dec. 1967.

Moses, J. (1971a): "Symbolic Integration: The Stormy Decade", Comm. of the ACM, vol. 14, pp. 548-560.

Moses, J. (1971b): "Algebraic Simplification A Guide for the perplexed", Comm. of the ACM vol. 14, pp. 527-537.

Moses, J. (1974): "The Evolution of Algebraic Manipulation Algorithms", Information Processing 1974, North-Holland Publ. Comp. pp. 483-488.

Murphy, G.M. (1960): "Ordinary Differential Equations", D. van Nostrand Comp., Inc. Princeton, New Jersey.

Richardson, D. (1968): "Some Undecidable Problems Involving Elementary Functions of a Real Variable" , The J. of Symb. Logic, vol. 33, pp. 514-520.

Schmidt, P. (1976): "Maschinelle symbolische Lösung von Differentialgleichungen 1. Ordnung und 1. Grades", Informatik Berichte, Universität Bonn, to appear.

Slagle, J.R. (1963): "A Heuristic Program that Solves Symbolic Integration Problems in Freshman Calculus", J. of the ACM, vol. 10, pp. 507-520 .

Spiegel, M.R. (1958): "Applied Differential Equations", Englewoog Cliffs, N.J., Prentice Hall, Inc.

Wang, P.S. (1971): "Evaluation of Definite Integrals by Symbolic Manipulation", Report MAC TR-92, poject MAC, MIT, Cambr., Mass.

Projective Representations for Energy Band Calculations

Ray Head
Computer Center, University of Arkansas

Introduction

For symmetry considerations in energy band calculations one needs to compute the irreducible representations of the group of the wave vector at a point inside or on the surface of the Brillouin zone. These symmetry considerations not only provide an effective mathematical tool for labeling the eigenvalues obtained but also enhance convergence, thereby reducing computer time required for the calculation. The geometry of the crystal provides us with a 2-cocycle ω of the associated point group G and the irreducible representations of the group of the wave vector can be very easily obtained from projective representations of G having the factor set ω. In the large majority of cases it can be shown that the group theoretical structure of G is such that the needed projective representations are all monomial, i.e. they can be induced from one-dimensional projective representations on certain subgroups of G. For these cases we present an inductive type algorithm which can be easily programmed for the actual computation.

This paper is then divided into two sections. In Section I we more precisely state the relation between the irreducible representations of the group of the wave vector and the projective representations of the associated point group and also develop needed machinery from the theory of projective representations. The algorithm is presented in Section II and the theorems required for its formulation are proved.

I. Irreducible representations of the group of the wave vector

Let k be a point lying inside or on the surface of the Brillouin zone. The group of the wave vector $\overline{G}$ for k consists of all those elements from the space group of the crystal whose rotational parts leave k essentially unchanged. That is, if $(g|a)$ is an element of $\overline{G}$ then $gk = k + X$ for some reciprocal lattice vector X. The rotational parts g of the elements $(g|a)$ form a group called the associated point group. Our task is to find all the irreducible representations T of $\overline{G}$ which satisfy the condition $T((e|t)) = e^{i(k,t)}I$ for any primitive translation vector t. For descriptions of how these representations are used in various band calculations we refer the reader to 2, 12, and 14 in the references.

Lyubarskii has provided a method of obtaining the irreducible representations of $\overline{G}$ from projective representations of the associated point group. This method is presented in Theorem 1. The remainder of this section is then devoted to a development of the theory needed to compute the projective representations and we begin with the following definitions. G will denote a finite group, K a field, and GL(M) the general linear group of a vector space M over K. K^* will denote the multiplicative group of non-zero elements of K.

Definition 1: Let $\omega: G \times G \to K^*$ satisfy $\omega(x,yz)\omega(y,z) = \omega(xy,z)\omega(x,y)$ for all x, y, z in G and $\omega(e,x) = \omega(x,e) = 1$ for all x in G. The map is then called a 2-cocycle or factor set. If there is a map $\phi: G \to K^*$ such that $\omega(x,y) = \phi(x)\phi(y)/\phi(xy)$ for all x, y in G, ω is called a 2-coboundary or splitting factor set. The group of all cocycles will be denoted $Z^2(G,K^*)$ and its subgroup of coboundaries $B^2(G,K^*)$. The quotient group $Z^2(G,K^*)/B^2(G,K^*)$ is the Schur multiplier $H^2(G,K^*)$. Two cocycles are said to be cohomologous if they lie in the same class of $Z^2(G,K^*)$ modulo $B^2(G,K^*)$.

Definition 2: The map $T: G \to GL(M)$ is called a projective representation of G having factor set ω if there is a map $\omega: G \times G \to K^*$ such that $T(e) = I$ and $T(x)T(y) = \omega(x,y)T(xy)$ for all x, y in G. T will be more briefly called an ω-rep of G. The degree of T is the dimension of M over K. Two projective representations R and S of G are said to projectively equivalent if there is a map $\phi: G \to K^*$ and a non-singular linear transformation B such that $R(g) = \phi(g)B^{-1}S(g)B$ for all g in G. If ϕ is identically 1, R and S are linearly equivalent.

Essentially all the proof of Theorem 1 may be found in Lyubarskii,[8] pages 94-95. In Theorem 1 K will be the field of complex numbers.

Theorem 1: Let $\overline{G}$ be the group of the wave vector for the point k and let G be the associated point group. The map ω on G X G defined by $\omega(g,h) = e^{i(g^{-1}k-k,b)}$, where $(g|a)$ and $(h|b)$ are elements of $\overline{G}$, is an element of $Z^2(G,K^*)$. Furthermore, if the representation S of $\overline{G}$ satisfies the condition $S((e|t)) = e^{i(k,t)}I$ for all primitive translations t, then there is a projective representation T of G having factor set ω such that $S((g|a)) = e^{i(k,a)}T(g)$ for all elements $(g|a)$ in $\overline{G}$.

Note that if k is inside the Brillouin zone we must have $gk = k$ for all g in G. Thus ω is

trivial and the problem is reduced to finding the irreducible (ordinary) representations of G. In most cases where k is on the surface, however, ω is a nonbounding cocycle.

Definition 3: Let ω be an element of $Z^2(G,K^*)$ and $\{a_g\}$ a set of elements in one-to-one correspondence with the elements of G. Consider the set of all formal sums of the form $\sum_g \beta_g a_g$ where β_g is an element of K. We construct an algebra over K, called the ω-group algebra ωKG, by defining

(1) $\sum_g \beta_g a_g + \sum_g \rho_g a_g = \sum_g (\beta_g + \rho_g) a_g$,

(2) $\alpha(\sum_g \beta_g a_g) = \sum_g (\alpha\beta_g) a_g$, and

(3) $(\sum_g \beta_g a_g)(\sum_g \rho_g a_g) = \sum_t \gamma_t a_t$ where $\gamma_t = \sum_g \beta_g \rho_{g^{-1}t} \omega(g, g^{-1}t)$.

Note that for g, h in G we have $a_g a_h = \omega(g,h) a_{gh}$.

A module over the algebra ωKG is called an ωKG-module (unless otherwise stated our modules are left modules). If $T: G \to GL(M)$ is an ω-rep of G then M is an ωKG-module and T is called the ω-rep of G afforded by M. Every ωKG-module affords an ω-rep of G.

Finally, let T be the ω-rep of G afforded by the ωKG-module M. The map $\phi: G \to K$ given by $\phi(g) = \mathrm{tr}(T(g))$ is called the character of G afforded by M.

Definition 4: Let M be an ωKG-module and N a K-subspace of M. N is called an ωKG-subspace if $a_g n$ is an element of N for every n in N. An ωKG-module which has no proper ωKG-subspace is said to be an irreducible ωKG-module. Otherwise the module is reducible. A completely reducible ωKG-module is the direct sum of irreducible ωKG-subspaces.

These properties of ωKG-modules correspond naturally to certain properties of the ω-reps of G which they afford. If T is the ω-rep of G afforded by the completely reducible ωKG-module M, then T can be written in the form

$$T(g) = \begin{bmatrix} T_1(g) & 0 \\ 0 & T_2(g) \end{bmatrix}$$ with a suitable choice of basis. A reducible ω-rep T can be written in the form

$$T(g) = \begin{bmatrix} T_1(g) & T_2(g) \\ 0 & T_3(g) \end{bmatrix}$$ and a ω-rep of G which cannot be written in this manner is irreducible.

Let us reiterate that we are trying to obtain certain projective representations of a finite group that have exactly the same factor set. Hence when we say that two ω-reps are equivalent we mean that they are linearly equivalent.

We now present several theorems which will be needed for the algorithm in Section II. Proofs for these theorems may be filled in utilizing techniques from the theory of ordinary representations. To acquaint the reader with the type of conversion necessary, the proof of Theorem 2, which is the projective analog of Maschke's Theorem, is given. This proof was adapted from a proof for the ordinary case in Curtis and Reiner,[4] pages 41-42.

Theorem 2: Suppose that char(K) does not divide the order of G. If M is any ωKG-module, M is completely reducible.

Proof: Let T be the ω-rep of G afforded by M and let N be an ωKG-subspace of M. Let $P: M \to N$ be the projection and define Q in $\mathrm{HOM}_K(M,M)$ by

$$Q = \frac{1}{|G|} \sum_g \frac{1}{\omega(g^{-1},g)} T(g) P T(g^{-1}).$$ Suppose that x is any element of G and consider T(x)Q. We have

$$T(x)Q = \frac{1}{|G|} \sum_g \frac{1}{\omega(g^{-1},g)} T(x)T(g)PT(g^{-1})$$

$$= \frac{1}{|G|} \sum_g \frac{\omega(x,g)}{\omega(g^{-1},g)} T(xg)PT(g^{-1}).$$

Set $h = xg$. Then $g^{-1} = h^{-1}x$ and hence

$$T(x)Q = \frac{1}{|G|} \sum_h \frac{\omega(x,x^{-1}h)}{\omega(h^{-1}x,x^{-1}h)} T(h)PT(h^{-1}x) =$$

$$\frac{1}{|G|} \sum_h \frac{\omega(x,x^{-1}h)}{\omega(h^{-1}x,x^{-1}h)\omega(h^{-1},x)} T(h)PT(h^{-1})T(x).$$

But $\omega(h^{-1}x,x^{-1}h)\ \omega(h^{-1},x) = \omega(h^{-1},h)\omega(x,x^{-1}h)$ so

$$T(x)Q = \frac{1}{|G|} \sum_g \frac{1}{\omega(g^{-1},g)} T(g)PT(g^{-1})T(x) = QT(x).$$

Note also that for any m in M and x in G, $\frac{1}{\omega(x^{-1},x)} T(x)PT(x^{-1})m = T(x)PT(x)^{-1}m$. Since N is an ωKG-subspace and $PT(x^{-1})m$ is an element of N we see that $\frac{1}{\omega(x^{-1},x)} T(x)PT(x^{-1})m$ is in N. Therefore Q is an element of $\mathrm{HOM}_K(M,N)$. Clearly $Qn = n$ for any element n of N so that Q is a projection of M onto N. Letting $L = \{m - Qm : m \in M\}$ we see from $T(x)Q = QT(x)$ for all x in G that L is also an ωKG-subspace of M. Because $M = N \oplus L$, M is completely reducible.

Corollary: Suppose that char(K) does not divide the order of G and let ω be any element of $Z^2(G,K^*)$. Then ωKG is semi-simple.

Now let T be the ω-rep of G afforded by the ωKG-module M. From Theorem 2 we see that there is an element A in GL(M) such that

$$A^{-1}T(g)A = \begin{bmatrix} T_1(g) & & 0 \\ & \ddots & \\ 0 & & T_n(g) \end{bmatrix}$$ for all g in G.

From now on we will assume K is the field of complex numbers so that Theorem 2 will be valid.

Definition 5: The T_i appearing in the above expansion are called the irreducible components of T. The number of T_j in this expansion equivalent to T_i is called the multiplicity of T_i in T. Equivalent definitions can be made concerning the irreducible ωKG-modules M_i which afford the ω-reps T_i and also about the irreducible characters afforded by the M_i.

The corollary to Theorem 2 allows us to use theorems concerning more general algebras. These theorems may be found in Curtis and Reiner,[4] pages 179-189, and Huppert,[5] pages 469-474. Their implications concerning ωKG-algebras are given in Theorems 3, 4, and 5.

Theorem 3. Let S and T be irreducible ω-reps of G and let B be a linear transformation such that $BT(g) = S(g)B$ for all g in G. Then $B = 0$ if S and T are inequivalent and $B = \alpha I$ for some α in K^* if

S and T are equivalent.

Theorem 4: Suppose that M and N are ωKG-modules affording the characters ϕ and ψ, respectively. Then $M \cong N$ if and only if $\phi = \psi$.

Theorem 5: Let $\{T_i\}_{i=1}^{s}$ be a complete set of inquivalent irreducible ω-reps of G and suppose deg $(T_i) = n_i$. Let n be the dimension of $Z(\omega KG)$ over K. Then $n = s$ and $\sum_{i=1}^{n} n_i^2 = |G|$.

Theorem 6: Suppose that M and N are irreducible ωKG-modules and let m be the dimension of M over K. Set $\bar{\delta} = 0$ if M and N are not isomorphic and $\bar{\delta} = 1$ otherwise.

(A) Let T and R be the ω-reps of G afforded by M and N, respectively, with $T(g) = (\alpha_{ij}(g))$ and $R(g) = (\beta_{ij}(g))$. Then

$$\frac{m}{|G|}\sum_g \frac{1}{\omega(g^{-1},g)}\alpha_{pr}(g)\beta_{st}(g^{-1}) = \bar{\delta}\delta_{pt}\delta_{rs}.$$

(B) Let ψ and ρ be the characters of G afforded by M and N, respectively. Then

$$\frac{1}{|G|}\sum_g \frac{1}{\omega(g^{-1},g)}\psi(g)\rho(g^{-1}) = \bar{\delta}.$$

Definition 6: Let ω be an element of $Z^2(G,K^*)$. A map $f:G\to K$ is called an ω-class function on G if $\frac{\omega(x^{-1},gx)\omega(g,x)}{\omega(x^{-1},x)} f(x^{-1}gx) = f(g)$ for all x, g in G. The set of all ω-class functions on G will be denoted by $C\omega(G)$.

Note that if ψ is the character of G afforded by some ωKG-module M, then ψ is in $C\omega(G)$.

Theorem 7: $C\omega(G)$ is a Hilbert space with inner product given by $(f,h) = \frac{1}{|G|}\sum_g f(g)\overline{h(g)}$ and the set of all irreducible characters of G forms an orthonormal basis. If ψ is any character of G and ϕ is irreducible, the multiplicity of ϕ in ψ is $(\psi,\phi)_G = \frac{1}{|G|}\sum_g \frac{1}{\omega(g^{-1},g)}\psi(g)\phi(g^{-1})$. If ψ is any character of G, ψ is irreducible if and only if $(\psi,\psi)_G = 1$.

Let H be a subgroup of G and ω an element of $Z^2(G,K^*)$. By an ω-rep of H we mean a ω_H-rep of H where ω_H denotes the restriction of ω to H.

Definition 7: Let T be the ω-rep of H afforded by the ωKH-module M and let $M^G = \omega KG \otimes_{\omega KH} M$. M^G becomes an ωKG-module by defining $a_g(a_t \otimes m) = (a_g a_t) \otimes m$ and is called the ωKG-module induced from M. The ω-rep of G afforded by M^G is called the ω-rep of G induced from T, T^G.

Suppose that $\{x_i\}_{i=1}^{n}$ is a complete set of left coset representatives of H in G and $\{m_j\}_{j=1}^{k}$ is a basis for M. Then $\{a_{x_i} \otimes m_j\}_{i=1,n\ :\ j=1,k}$ is a basis of M^G and therefore $\deg(T^G) = nk$. Let g be any element of G and suppose that $gx_i = x_j h$ for some h in H. For any m in M we have $a_g(a_{x_i} \otimes m) = (a_g a_{x_i}) \otimes m = \omega(g,x_i)a_{gx_i} \otimes m = \omega(g,x_i)a_{x_j h} \otimes m = \frac{\omega(g,x_i)}{\omega(x_j,h)} a_{x_j}a_h \otimes m =$ $a_{x_j} \otimes \frac{\omega(g,x_i)}{\omega(x_j,h)} a_h m$. Since $h = x_j^{-1}gx_i$,

$$\frac{\omega(g,x_i)}{\omega(x_j,h)} = \frac{\omega(g,x_i)}{\omega(x_j,x_j^{-1}gx_i)} = \frac{\omega(x_j^{-1},gx_i)\omega(g,x_i)}{\omega(x_j^{-1},x_j)}$$

and therefore $a_g(a_{x_i} \otimes m) = a_{x_j} \otimes \frac{\omega(x_j^{-1},gx_i)\omega(g,x_i)}{\omega(x_j^{-1},x_j)} a_{x_j^{-1}gx_i} m$. If we let $T_o(g) = T(g)$ if g is in H and $T_o(g) = 0$ otherwise, then $T^G(g)$ can be written as a matrix whose i,jth block is

$$\frac{\omega(x_j^{-1},gx_i)\omega(g,x_i)}{\omega(x_j^{-1},x_j)} T_o(x_j^{-1}gx_i).$$

Theorem 8: Suppose that L and H are subgroups of G with $L \leq H$. For any ω-rep T of L, $(T^H)^G$ and T^G are equivalent.

Theorem 9: Let $H \leq G$ and ϕ the character of H afforded by the ωKH-module M. Then the character of G afforded by M^G, denoted ϕ^G, is given by $\phi^G(g) = \frac{1}{|H|}\sum_{x\in G} \frac{\omega(x^{-1},gx)\omega(g,x)}{\omega(x^{-1},x)} \phi_o(x^{-1}gx)$, where $\phi_o(t) = \phi(t)$ if t is in H and $\phi_o(t) = 0$ otherwise.

Theorem 10: Let $H \leq G$, ϕ an irreducible character of H, and ψ an irreducible character of G. Then $(\phi^G,\psi)_G = (\psi_H,\phi)_H$.

Definition 8: Let $H \Delta G$, g an element of G, and M an ωKH-module. We define a new action of ωKH on the vector space M by setting $a_h * m = \left[\frac{\omega(g^{-1},hg)\omega(h,g)}{\omega(g^{-1},g)} a_{g^{-1}hg}\right]m$. The ωKH-module M^g with this operation is called a G-conjugate of M. The ω-reps of H afforded by conjugate ωKH-modules are also called conjugate. The same terminology and notation will be applied to characters.

Note that if T is the ω-rep of H afforded by M, then the ω-rep T^g afforded by M^g is given by $T^g(h) = \frac{\omega(g^{-1},hg)\omega(h,g)}{\omega(g^{-1},g)} T(g^{-1}hg)$.

Theorem 11: Suppose that M is an irreducible ωKG-module and $H \Delta G$. The irreducible components of M_H are all conjugate.

Definition 9: Let $H \Delta G$, M an irreducible ωKH-module, and ϕ the character of H afforded by M. We set $G\phi = \{g \in G : \phi^g = \phi\}$ and call $G\phi$ the little ω-group of ϕ in G.

Observe that $H \Delta G\phi$ and $[G:G\phi]$ is the number of distinct G-conjugates of ϕ.

Theorem 12: Let $H \Delta G$, M an irreducible ωKH-module, and T the ω-rep of H afforded by M. Then T^G is irreducible if and only if for all x not in H, T^x is not equivalent to T.

II. The algorithm

In this section we present a method for finding most of the projective representations mentioned in Theorem 1. The theorems on which the algorithm is based are generalizations of theorems proved by Talburt[13] for the computation of irreducible (ordinary) representations of metabelian groups. We will be able to accomplish the same goal for ω-reps of strongly metabelian groups. Using our results and those of Talburt we can then compute the representations at all points inside the Brillouin zone, at all points on the surface where the 2-cocycle ω is a coboundary, and at all points on the surface where the associated point group is strongly metabelian.

Definition 10: The group G is called strongly metabelian if it has a cyclic normal subgroup H such that G/H is abelian.

Theorem 13: Let p be a prime, $H \Delta G$, $[G:H] = p$, and T a one-dimensional ω-rep of H. If there is an x not in H such that $T^x = T$, then T can be extended to p one-dimensional ω-reps of G.

Proof: If there is an x not in H such that $T^x = T$ then the little ω-group of T is G, that is $T^y = T$ for all y in G. Let $G = H \cup xH \ldots \cup x^{p-1}H$ be a left coset decomposition of H in G and suppose that $y = x^r h$ is any element of G. Let v

be a pth root of $T(x^p)\prod_{i=1}^{p-1}\omega(x,x^i)$ and define $\overline{T}$ on G by

$$\overline{T}(y) = \begin{cases} T(h) & \text{if } r = 0 \\ \dfrac{v^r T(h)}{\omega(x^r,h)\prod_{i=0}^{r-1}\omega(x,x^i)} & \text{if } r > 0. \end{cases}$$

To show that $\overline{T}$ is well-defined, assume that $y = x^r h = x^s k$ where $s = r + mp$ and h, k are in H. Then $h = x^{s-r}k = x^{mp}k$. Using $y = x^s k$ we have $\overline{T}(y) = \dfrac{v^s T(k)}{\omega(x^s,k)\prod_{i=0}^{s-1}\omega(x,x^i)}$ for $s > 0$ and we can clearly assume that this is the case. If $m = 0$ there is nothing to prove so we also assume $m > 0$.

Then $\overline{T}(y) = \dfrac{v^r v^{mp} T(k)}{\omega(x^s,k)\prod_{i=0}^{s-1}\omega(x,x^i)} =$

$$\frac{v^r (T(x^p))^m \left(\prod_{i=1}^{p-1}\omega(x,x^i)\right)^m T(k)}{\omega(x^s,k)\prod_{i=0}^{s-1}\omega(x,x^i)} =$$

$$\frac{v^r T(x^{pm})\prod_{i=0}^{m-1}\omega(x^p,x^{pi})\prod_{i=1}^{p-1}\omega(x,x^i)^m T(k)}{\omega(x^s,k)\prod_{i=0}^{s-1}\omega(x,x^i)} =$$

$$\frac{v^r T(h)\omega(x^{pm},k)\prod_{i=0}^{m-1}\omega(x^p,x^{pi})\prod_{i=1}^{p-1}\omega(x,x^i)^m}{\omega(x^s,k)\prod_{i=0}^{s-1}\omega(x,x^i)}.$$

Now $\omega(x^r,h)\omega(x^{pm},k) = \omega(x^r,x^{-r}x^s k)\omega(x^{-r}x^s,k) =$

$\omega(x^s,k)\omega(x^r,x^{pm})$ so that $\omega(x^{pm},k) =$

$\dfrac{\omega(x^s,k)\omega(x^r,x^{pm})}{\omega(x^r,h)}$ and hence

$$\overline{T}(y) = \frac{v^r T(h)\omega(x^r,x^{pm})\prod_{i=0}^{m-1}\omega(x^p,x^{pi})\prod_{i=1}^{p-1}\omega(x,x^i)^m}{\omega(x^r,h)\prod_{i=0}^{s-1}\omega(x,x^i)}$$

Observing that for $0 \le q \le r-1$ we have $\omega(x^{r-q},x^{s-r}) =$

$\dfrac{\omega(x,x^{r-q-1}x^{s-r})\omega(x^{r-q-1},x^{s-r})}{\omega(x,x^{r-q-1})} =$

$\dfrac{\omega(x,x^{s-q-1})\omega(x^{r-q-1},x^{s-r})}{\omega(x,x^{r-q-1})}$, we can easily obtain the result $\omega(x^r,x^{mp}) = \dfrac{\prod_{i=s-r}^{s-1}\omega(x,x^i)}{\prod_{i=0}^{r-1}\omega(x,x^i)}$. By induction on m we can show that $\prod_{i=0}^{m-1}\omega(x^p,x^{pi})\prod_{i=1}^{p-1}\omega(x,x^i)^m = \prod_{i=1}^{mp-1}\omega(x,x^i)$. This is clearly true for $m = 1$ so assume that $m > 1$ and $\prod_{i=0}^{m-2}\omega(x^p,x^{pi})\prod_{i-1}^{p-1}\omega(x,x^i)^{m-1} =$

$\prod_{i=1}^{P(m-1)-1}\omega(x,x^i)$. Now for $0 \le q \le p-1$ we have

$\omega(x^{p-q},x^{pm-p})\omega(x,x^{p-q-1}) = \omega(x^{p-q-1},x^{pm-p})$

$\omega(x,x^{pm-q-1})$ so that $\omega(x^p,x^{pm-p})\prod_{i=1}^{p-1}\omega(x,x^i) =$

$\prod_{i=pm-p}^{pm-1}\omega(x,x^i)$. Therefore $\prod_{i=0}^{m-1}\omega(x^p,x^{pi})\prod_{i=1}^{p-1}\omega(x,x^i)^m$

$$= \omega(x^p,x^{pm-p})\prod_{i=1}^{p-1}\omega(x,x^i)\prod_{i=0}^{m-2}\omega(x^p,x^{pi})\prod_{i=1}^{p-1}\omega(x,x^i)^{m-1}$$

$$= \prod_{i=1}^{pm-p-1}\omega(x,x^i)\prod_{i=pm-p}^{pm-1}\omega(x,x^i) = \prod_{i=1}^{pm-1}\omega(x,x^i).$$

Thus $\overline{T}(y) = \dfrac{v^r T(h)\dfrac{\prod_{i=s-r}^{s-1}\omega(x,x^i)}{\prod_{i=0}^{r-1}\omega(x,x^i)}\prod_{i=1}^{mp-1}\omega(x,x^i)}{\omega(x^r,h)\prod_{i=0}^{s-1}\omega(x,x^i)}$

$$= \frac{v^r T(h)}{\omega(x^r,h)\prod_{i=0}^{r-1}\omega(x,x^i)}$$ and $\overline{T}$ is well defined.

To show that $\overline{T}$ is an ω-rep of G we will use the following three equalities easily verified using the definition:

(1) $\omega(x^r,h)\overline{T}(x^r h) = \overline{T}(x^r)\overline{T}(h)$,

(2) $\omega(x^r,x^s)\overline{T}(x^r x^s) = \overline{T}(x^r)\overline{T}(x^s)$, and

(3) $\omega(x^r h,k)\overline{T}(x^r hk) = \overline{T}(x^r h)\overline{T}(k)$.

Now consider $(x^r h)(x^s k) = x^r x^s x^{-s} h x^s k$. By (3) we have $\overline{T}(x^r h x^s k)\omega(x^r x^s x^{-s} h x^s,k) = \overline{T}(x^r x^s x^{-s} h x^s)\overline{T}(k)$. Thus $\overline{T}(x^r h x^s k)\omega(x^r h x^s,k)\omega(x^r x^s,x^{-s} h x^s) =$ $\overline{T}(x^r x^s)\overline{T}(x^{-s} h x^s)\overline{T}(k)$ and hence $\overline{T}(x^r h x^s k)\omega(x^r h x^s,k)\omega(x^r,x^s)\omega(x^r x^s,x^{-s} h x^s) =$ $\overline{T}(x^r)\overline{T}(x^s)\overline{T}(x^{-s} h x^s)\overline{T}(k)$. This implies that

$\overline{T}(x^r h x^s k)\ AB = \overline{T}(x^r)\overline{T}(h)\overline{T}(x^s)\overline{T}(k) =$

$\overline{T}(x^r h)\overline{T}(x^s k)$ where

$A = \omega(x^r h x^s,k)\omega(x^r,x^s)\omega(x^r x^s,x^{-s} h x^s)$ and

$$B = \frac{\omega(x^{-s},hx^s)\omega(h,x^s)}{\omega(x^r,h)\omega(x^s,k)\omega(x^{-s},x^s)}$$

But we find $AB = \dfrac{\omega(x^r h x^s,k)\omega(x^r,hx^s)\omega(h,x^s)}{\omega(x^r,h)\omega(x^s,k)} =$

$\dfrac{\omega(x^r h x^s,k)\omega(x^r h,x^s)}{\omega(x^s,k)} = \omega(x^r h,x^s k)$ so that

$\overline{T}$ is an ω-rep of G. This completes the proof of Theorem 13.

Theorem 14: Let G be a strongly metabelian group, H and $\overline{H}$ normal subgroups of G with $\overline{H}$ cyclic, $G/\overline{H}$ abelian, and $[G:H] = p$. Suppose that $\{T_j\}$ is a family of one-dimensional ω-reps of certain subgroups $H_j \le H$ such that (1) any irreducible ω-rep of H is equivalent to T_j^H for some j, and (2) $\overline{H} \le H_j$ for all j.

Then given any irreducible ω-rep T of G either $T \sim T_j^G$ for some j or $T \sim \overline{T}^G$ where $\overline{T}$ is an extension of T_j to a one-dimensional ω-rep of a subgroup $\overline{H}_j \leq G$ which contains H_j as a normal subgroup of index p.

Proof: Let T be any irreducible ω-rep of G and suppose that T_H is irreducible. Then $T_H \sim T_j^H$ for some j. For any x not in H, $T_H^x(h) = \frac{\omega(x^{-1},hx)\omega(h,x)}{\omega(x^{-1},x)} T_H(x^{-1}hx) = \frac{\omega(x^{-1},hx)\omega(h,x)}{\omega(x^{-1},x)}$

$T(x^{-1}hx) = T(x)^{-1}T_H(h)T(x)$ and therefore $T_H^x \sim T_H$. Thus $(T_H)^G$ is not irreducible. Since $(T_H)^G \sim (T_j^H)^G \sim T_j^G$, it follows that T_j^G is not irreducible. So there exists an x not in H_j such that $T_j^x = T_j$. Since T_j^H is irreducible, x is not in H. But x^p is in H and $T_j^{x^p} = T_j$ which implies x^p is in H_j. We let $\overline{H}_j = H_j \cup xH_j \cup \ldots \cup x^{p-1}H_j$. Then $\overline{H}_j$ is a normal subgroup of G containing H_j as a normal subgroup of index p. Because $\overline{H}_j \cap H = H_j$ we have $p = [\overline{H}_j:H_j] = [\overline{H}_j:\overline{H}_j \cap H] = [H\overline{H}_j:H]$ and therefore $G = H\overline{H}_j$.

If $\{x_i\}_{i=1}^n$ is a complete set of left coset representatives of H_j in H and g is not in $\overline{H}_j$, then $g = ht$ where h is in H and t is not in $\overline{H}_j$. Suppose that $h = x_i k$ for some k in H_j. Then $g = x_i kt$ where kt is in $\overline{H}_j$. If $x_i x_j^{-1}$ is in $\overline{H}_j$, then $x_i x_j^{-1} = h$ for some h in $\overline{H}_j$ and hence $x_i x_j^{-1}$ is in $H \cap \overline{H}_j = H_j$. This is impossible so $\{x_i\}_{i=1}^n$ is a complete set of left coset representatives of $\overline{H}_j$ in G. It follows that if $\overline{T}$ is one of the p extensions of T_j to $\overline{H}_j$, then $(\overline{T}^G)_H = T_j^H \sim T_H$.

Now let x be in G, x not in $\overline{H}_j$ and assume that $\overline{T}^x = \overline{T}$. Then we would have $\overline{T}^{x_i} = \overline{T}$ for some i implying that $T_j^{x_i} = T_j$. This would contradict T_j^H being irreducible. Thus $\overline{T}^G$ is irreducible. Suppose that two of the extensions, say $\overline{T}_1$ and $\overline{T}_2$, induce to equivalent ω-reps of G. Then we would have $\overline{T}_1 = \overline{T}_2^x$ for some x not in $\overline{H}_j$. But $\overline{T}_2^x = \overline{T}_2^{x_i}$ for some i and again $T_j = T_j^{x_i}$.

Therefore the p extensions of T_j to $\overline{H}_j$ induce to p inequivalent irreducible ω-reps of G, all of which have the same degree as T_j^H and all of which are irreducible components of $(T_H)^G$. But $\deg((T_H)^G) = p \deg(T_H)$ so that $(T_H)^G$ can have at most p irreducible components of degree equal to $\deg(T_H)$. Since T is also an irreducible component of $(T_H)^G$, it follows that $T \sim \overline{T}^G$ for one of the extensions.

Now assume that T_H is reducible and let R be an irreducible component of T_H. First suppose that R^G is not irreducible. Using the arguments above we see that R^G has exactly p irreducible components and all of these components have the same degree as R. This is impossible since T is also an irreducible component of R^G and we have assumed that T_H is reducible. Thus R^G is irreducible and $R^G \sim T$. Therefore $T \sim T_j^G$ for some j and we are finished.

Talburt's theorem for ordinary representations is also important for our purposes and we will give it here.

Theorem 15. Let $H \Delta G$, $[G:H] = p$, and $\{T_j\}$ be a family of linear representations on certain subgroups $H_j \leq H$ such that (1) any irreducible representation of H is equivalent to T_j^H for some j, and (2) $H_j \Delta G$ for all j.

Then given any irreducible representation T of G, either $T \sim T_j^G$ for some j or there is a subgroup $\overline{H}_j \leq G$ and an extension $\overline{T}$ of T_j to a linear representation of $\overline{H}_j$ such that $[\overline{H}_j:H_j] = p$ and $T \sim \overline{T}^G$.

Talburt applied this theorem to metabelian groups and constructed the irreducible representations inductively. Let G be metabelian and $\{e\} = K_0 \leq K_1 \leq \ldots \leq K_n = G$ a solvable series for G with $K_j \Delta K_{j+1}$ and the index p_{j+1} of K_j in K_{j+1} prime. Suppose that K_t is an abelian normal subgroup of G with G/K_t abelian. We can then apply Theorem 15 to K_{t+1} using the linear representations of K_t as the initial inducing family. These linear representations will be prolonged as far as possible up the solvable series and finally induced to G.

The same procedure can be used in the projective case for strongly metabelian groups using Theorem 14 and the following two theorems. Theorem 16 is proved by Huppert,[5] page 643.

Theorem 16: If G is cyclic, $H^2(G,K^*) = \{e\}$. That is, every 2-cocycle is a coboundary.

Theorem 17: Let ω, ρ be in $Z^2(G,K^*)$. Then ω is a coboundary if and only if G has a one-dimensional ω-rep. If $\phi:G \to K^*$ with $\omega(g,h) = \frac{\phi(g)\phi(h)}{\phi(gh)} \rho(g,h)$ for all g, h in G and R is any irreducible ρ-rep of G, then T defined on G by $T(g) = \phi(g)R(g)$ is an irreducible ω-rep of G. All the irreducible ω-reps of G can be obtained in this manner.

Now let G be strongly metabelian with $\{e\} = K_0 \leq K_1 \leq K_2 \leq \ldots \leq K_n = G$ a solvable series for G. This time some K_t is a cyclic normal subgroup of

G with G/K_t abelian. From Theorem 16 we see that for $j \leq t$, ω_{K_j} is a coboundary and hence by Theorem 17 the irreducible ω-reps of K_j are all one-dimensional. These may be found from the ω-reps of K_{j-1} using Theorem 13. The procedure in the projective case for strongly metabelian groups is then the same as in the ordinary case for metabelian groups. By these remarks we then have the following algorithm, which is presented for the projective case. We assume that for $1 \leq j \leq n$ there is a family of one-dimensional ω-reps $\{T_m\}_{m=1}^{n_j}$ defined on certain subgroups $\{L_m\}$ contained in K_{j-1} such that any irreducible ω-rep of K_{j-1} is linearly equivalent to an induced ω-rep $T_m{}^{H_{j-1}}$. We proceed to construct similar families $\{S\}$ and $\{N\}$ for K_j.

Computational Algorithm:

(0) Set $m = 1$.

(1) Choose an element g in K_j not in K_{j-1} which has not been used in (2) and (3). If all such elements have been exhausted set $S_{1m} = T_m$, $N_{1m} = L_m$ and go to (5).

(2) If g^{P_j} is not in L_m, return to (1).

(3) If $(\omega(g^{-1},hg)\omega(h,g)/\omega(g^{-1},g))T_m(g^{-1}hg) \neq T_m(h)$ for some h in L_m, return to (1).

(4) For each q, $1 \leq q \leq P_j$, let a_q be a P_jth root of $(T_m(g^{P_j})^{P_j^{-1}}\prod_{s=1}\omega(g,g^s))$ and define P_j one-dimensional ω-reps on the subgroup $N_{qm} = \langle L_m, g \rangle$ as follows. For $y = g^r h$ in N_{qm} set $S_{qm}(y) = T_m(h)$ if $r = 0$ and $S_{qm}(y) = a_q{}^r T_m(h)/\omega(g^r,h)\prod_{s=0}^{r-1}\omega(g,g^s)$ if $r > 0$.

(5) Set $m = m + 1$ and return to (1) if $m \leq n_j$.

As promised earlier in the section we will be able to use these methods to obtain the irreducible representations for the group of the wave vector at all points inside the Brillouin zone and at most points on its surface. Let's first consider those points inside. Here the cocycle for the associated point group is identically 1 and the problem reduces to finding the irreducible ordinary representations of the associated point group. For these points we can use Theorem 15 because of the thirty-two point groups there are (up to isomorphism) only two which are not metabelian. These are S_4, the symmetric group of degree 4, and S_4 X C_2 where C_2 denotes the cyclic group of order 2.

The alternating group of degree 4 is, however, metabelian and hence all of its irreducible representations can be induced from one-dimensional representations on subgroups normal in A_4. In fact the abelian normal subgroup F of A_4 which has abelian quotient is isomorphic to the Klein 4-group and contains all those permutations in S_4 which can be written as the product of two transpositions. Hence F is also normal in S_4. All of the irreducible representations of A_4 can therefore be obtained by extending a one-dimensional representation of F to A_4 or inducing a one-dimensional representation of F to A_4. The one-dimensional extensions of these representations of F in S_4 are not, however, extensions to subgroups which are normal in S_4 so we cannot use Theorem 15 directly to obtain the irreducible representations of C_2 X S_4. But since there is an element in $Z(C_2 \text{ X } S_4)$ not in S_4 we easily see that we can apply the theorem even though these subgroups are not normal in C_2 X S_4. Therefore we are able to use our algorithm at all points inside the Brillouin zone and at all points on the surface of the Brillouin zone where the factor set of the associate point group is a coboundary.

Our success at other points on the surface of the Brillouin zone depends solely on the strongly metabelian nature of the associated point group. Of the thirty-two point groups there are (again up to isomorphism) four which are not strongly metabelian. These are A_4, S_4, C_2 X A_4, and C_2 X S_4, all of which only occur in the cubic symmetry.

Fortran IV programs to implement the above algorithm in solid state routines have been written by the author. These programs receive as input operations in the associated point group and their corresponding translations. After computing a group table and the 2-cocycle of Theorem 1, a trial and error routine is used to find a solvable series for the group. The paper by Jürgesen[6] has been found to be especially useful for this purpose. The group table resulting from this routine is then used in a subroutine based on the above algorithm to compute the projective representations. Energy band programs written by Keown, Wiff, and Smart at Texas A&M utilizing Slater's Augmented Plane Wave Method have been adapted to provide a suitable interface between the Group Theory and the Solid State Physics. Questions concerning programming are invited.

References

1. C. Brott and J. Neubüser. A programme for the calculation of characters and representations of finite groups. Computational Problems in Abstract Algebra (Pergamon Press, 1970).

2. J. F. Cornwell. Group Theory and Electronic Energy Bands in Solids (North Holland-J. Wiley, 1969).

3. H. S. M. Coxeter and W. O. J. Moser. Generators and Relations for Discrete Groups (Springer, 1965).

4. C. W. Curtis and I. Reiner. Representation Theory of Finite Groups and Associative Algebras (J. Wiley, 1962).

5. B. Huppert. Endliche Gruppen I (Springer, 1967).

6. H. Jürgesen. Calculation with the elements of a finite group given by generators and defining relations. Computational Problems in Abstract Algebra (Pergamon Press, 1970).

7. R. Keown. An Introduction to Group Representation Theory (Academic Press, 1975).

8. G. Lyubarskii. The Application of Group Theory in Physics (Pergamon Press, 1960).

9. S. C. Miller and W. F. Love. Tables of Irreducible Representations of Space Groups and Co-Representations of Magnetic Space Groups (Pruett Press, 1967).

10. P. G. Rudd and R. Keown. The computation of irreducible representations of finite groups of order 2^n, $n \leq 6$. Computational Problems in Abstract Algebra (Pergamon Press, 1970).

11. Gary M. Seitz. M-Groups and the Supersolvable Residual. Math Z., 110, 101-122 (1969).

12. J. S. Slater. Quantum Theory of Molecules and Solids, vol. 2 (McGraw-Hill Book Company, 1965).

13. John Talburt. Ph.D. Thesis, Department of Mathematics, University of Arkansas (1971, unpublished).

14. G. G. Wepfer, T. C. Collins, R. N. Euwema, and D. J. Stuckel. Symmetrization Techniques in Relativistic OPW Energy Band Calculations. Computational Methods in Band Theory (Plenum Press, 1971).

Perturbation Calculations for the Spin Up Problem Using Reduce

I. Cohen
Institute of Theoretical Physics, University of Stockholm,
S-113 46 Stockholm, Sweden

F. Bark
Department of Mechanics, Royal Institute of Technology
S-100 44 Stockholm, Sweden

Abstract

The algebraic computer language REDUCE is used to obtain expansions for the physical variables in the spin-up problem for a rapidly rotating gas at low Mach number.

1. Introduction

In recent years several papers have appeared in which algebraic systems have been used to obtain perturbation solutions for problems occuring in fluid mechanics, reference [1] is such an example. Perturbation techniques have become a basic tool of research workers in applied mathematics and in the past twenty or so years many different methods have been devised to combat the difficulties which often arise when using such methods [2]. For example consider the expansion of $\exp(-\epsilon t)$ which is discussed in [2].

$$e^{-\epsilon t} = \sum_{n=0}^{\infty} (-1)^n \frac{(\epsilon t)^n}{n!}$$

is uniformly convergent, but if we stop the series after a finite number of terms and choose a sufficiently large t so that $\epsilon t = 0(1)$ the truncated series will be an inaccurate approximation of $\exp(-\epsilon t)$. What one wishes to have is a truncated series which gives a satisfactory approximation for all values of t.

A straight-forward expansion is usually fairly simple to extend to high order and in some cases this might profitably be done using an automatic algebraic manipulation system [1]. However if the expansion is non-uniform most methods devised to eliminate the non-uniformity tend to increase the complexity of the calculations very rapidly. This is quite unfortunate as perturbation techniques have several advantages over the rival approximation method, numerical analysis. Assuming the expansion has been carried to a satisfactory high order one can often obtain more qualitative information from it, than that of a numerical solution. Then the former, because one works with exact arithmetic, can even give a higher precision than the latter, which is limited by the precision of the numerical techniques used, typical examples of this being boundary layer problems.

In the following we summarize an attempt which we made to obtain an expansion for some of the physical variables which appear in the spin-up problem for a rapidly rotating gas at low Mach number. Details may be found in [3].

All calculations were done using REDUCE and we seldom needed to intervene with a hand calculation. We chose REDUCE [4] as we feel that among the standard systems it is the most satisfactory for perturbation calculations. It is easy to learn, and has a pleasant syntax and many useful functions for this type of work and most important is available in an interactive environment. This latter is very important because of the trial and error nature of the work.

2. The Problem

One considers a viscous, thermally conducting gas which is rotating so rapidly that gravitational forces may be neglected in comparison with centrifugal forces. One also assumes a cylindrical symmetric region so that all variables only depend on the (scaled) cylindrical radius r and the time t. Then the change in pressure p due to a slight change in the rate of rotation satisfies the following equation

$$\frac{\epsilon r p_{rt} + \frac{4\epsilon}{\delta} p_t}{\sigma + \epsilon r^2} - \frac{1}{4r}\left[r \frac{\sigma p_{rt} - \frac{4\epsilon r}{\delta} p_t}{\sigma + \epsilon r^2} \right]_r$$

$$- \frac{1}{4r}\left[r \rho_0^{-\frac{1}{2}} \chi \left(p_r - \frac{4\epsilon(\delta+1) r p}{\sigma \delta} \right) \right]_r$$

$$= -\frac{1}{2r}\left[r^2 \chi \rho_0^{-\frac{1}{2}} \right]_r \qquad (1)$$

where

σ, δ are physical constants depending on the gas

$$\chi = (1 + \epsilon r^2)^{-\frac{3}{4}}$$

$$\rho_0 = \exp\left[\frac{(\delta+1)}{2\sigma\delta}(r^2 - 1)\right]$$

and ϵ is essentially the square of the Mach number.

From p the change in density ρ may be calculated

$$\rho = \frac{\epsilon r p_r + \frac{4\epsilon}{\delta} p}{(\sigma + \epsilon r^2)} \qquad (2)$$

and one can also obtain the change in temperature T and velocity from similar formulae.

There are two constraints on the solution for p, firstly the initial condition

$$p = 0 \text{ when } t = 0 \qquad (3)$$

also global conservation of mass requires

$$\int_0^1 r\rho\, dr = 0 \qquad (4)$$

3. A Straight-Forward Expansion

For small Mach numbers, ϵ is small and a perturbation expansion for p is a possibility

$$p = \overset{0}{p} + \epsilon \overset{1}{p} + \epsilon^2 \overset{2}{p} + 0(\epsilon^3)$$

On substituting this in (1) one obtains a set of equations for different orders of ϵ, which are to be solved starting with the lowest order

$$(r\overset{0}{p}_r)_r + (r\overset{0}{p}_{rt})_r = 4r$$

A solution of this for which (3) is satisfied is $r^2(1 - e^{-t})$, however $\overset{0}{p}$ must also satisfy (4). Assuming then that

$$\overset{0}{p} = r^2(1 - e^{-t}) + \overset{0}{f}(t)$$

then the integral is easily computed using (2) giving

$$\overset{0}{f}(t) = -\frac{(\delta+2)}{4}(1 - e^{-t})$$

Once $\overset{0}{p}$ has been calculated one can move on to the first order equation which has the form

$$(r\overset{1}{p}_r)_r + (r\overset{1}{p}_{rt})_r = \{\text{terms containing } r \text{ and } e^{-t}\}$$

A solution for $\overset{1}{p}$ can easily be obtained so that (3) is satisfied. Then assuming as before a correction to this $\overset{1}{f}(t)$, the integral (4) can be computed using (2) to finally give a $\overset{1}{p}$ which satisfies both (3) and and (4).

In fact to any order one has

$$(r\overset{n}{p}_r)_r + (r\overset{n}{p}_{rt})_r = \{\text{terms containing } r, t \text{ and } e^{-t}\}$$

The calculation of $\overset{n}{p}$ and of the correction terms $\overset{n}{f}(t)$ is completely algorithmic and the process may thus be continued to any order. We took the expansion of p to the second order and obtained values which are in very close agreement with a numerical program. Moreover inspection of the expansions obtained for ρ and the temperature T shows that ρ becomes negative near the centre of the container and positive near the periphery, and that T becomes negative everywhere. Both these conclusions are in agreement with numerical calculations.

Unfortunately the expansions obtained are not uniform as terms of the form te^{-t} occur in $\overset{1}{p}$ and t^2e^{-t} in $\overset{2}{p}$. This means that for $t = 0(\frac{1}{\epsilon})$ the truncated expansions are not satisfactory approximations.

4. Alternative Perturbation Expansions

Many methods have been devised to try and eliminate non-uniformity. The suitability of a given method depends, of course, on the problem under consideration.

We tried several methods the most important being the method of strained coordinates [2]. One assumes a coordinate transformation from (r, t) to (ℓ, s) so that

$$t = s + \epsilon \overset{1}{\nu}(s,\ell) + \epsilon^2 \overset{2}{\nu}(s,\ell) + 0(\epsilon^3)$$

$$r = \ell + \epsilon \overset{1}{\mu}(s,\ell) + \epsilon^2 \overset{2}{\mu}(s,\ell) + 0(\epsilon^3)$$

Then the derivatives $\frac{\partial}{\partial r}$ and $\frac{\partial}{\partial t}$ must be expressed in terms of $\frac{\partial}{\partial \ell}$ and $\frac{\partial}{\partial s}$ as well as the derivatives of $\overset{i}{\nu}$ and $\overset{i}{\mu}$ $(i = 1, 2)$. This means for example one cannot now write DF (P( ), R) for p_r in REDUCE. In fact special procedures for $\frac{\partial}{\partial r}$ and $\frac{\partial}{\partial t}$ must be constructed.

Assuming first that $\overset{i}{\mu} = 0$, one obtains the Poincaré-Lighthill-Kuo or PLK method. Here one quickly finds that because of the continuity equation (4) it seems impossible to obtain an expansion of the desired form. One runs into formidable coupled integro-differential equations for $\overset{0}{p}$ and $\overset{1}{\nu}$.

We next tried assuming that instead $\overset{i}{\nabla} = 0$, that is only stretching the "good" variable r. This was done in order to bypass the difficulties encountered with the continuity equation.

One quickly obtains $\overset{0}{p}$ as before using (4). Then $\overset{1}{\mu}$ is computed and one finds it is of the form

$$\overset{1}{\mu} = \overset{1}{A}\,\ell + \overset{2}{A}\,\ell^3$$

where $\overset{1}{A}$ and $\overset{2}{A}$ are essentially of the form $\frac{s}{e^s - 1}$.

This expression is well-behaved both for $s \to 0$ and $s \to \infty$. Finally to this order one obtains $\overset{1}{p}$ from (4).

The second order equation can now be calculated. It is found that $\overset{2}{\mu}$ contains a term of the form $s^2 e^{-s}$, and thus the expansion is not uniformly valid.

Another method attempted was the method of multiple scales [2]. Here one assumes different time scales. So that to first order $t_0 = t$, $t_1 = \epsilon t$ and thus

$$\frac{\partial}{\partial t} = \frac{\partial}{\partial t_0} + \epsilon \frac{\partial}{\partial t_1}$$

and $p = p(r, t_0, t_1)$. However we had no success with this method or several variants which we tried.

5. Conclusions

Although we were unsuccessful in obtaining a uniform expansion for the pressure p, the straight-forward expansion to second order can be used in practise for many gases and fairly large times t. In fact it proved to be a good check for a numerical program, where the gas taken was air.

That one cannot obtain a uniform expansion by any usual method, is a common feature of this field when dealing with real problems. However, the calculations performed in this investigation were of such a size that they would have been almost impossible to perform (accurately) by hand. The second order equations and solutions both for p and those used in investigating the method of strained coordinates, could only have been obtained, we believe, with an algebraic computer system. In fact it is unlikely that the calculation for $\overset{2}{\mu}$ would have been carried out by hand, and thus once upon a time one might have accepted the first order expansion $r = \ell + \epsilon \overset{1}{\mu}$ and taken it on trust that $\overset{2}{\mu} = \overset{1}{\mu}\, 0(1)$ for all t. It would appear that automatic symbol manipulation gives not only greater freedom but also greater responsibility.

6. References

1. R. W. Atherton and G. M. Homsy, J. of Comp. Physics 13, (1973), 45.

2. A. Nayfeh, "Perturbation Methods", John Wiley & Sons, New York, 1973.

3. I. Cohen and F. Bark, Royal Institute of Technology, Stockholm, Technical Report, TRITA-MEK-76-03.

4. A. C. Hearn, "Proceedings of the 2nd Symposium on Symbolic and Algebraic Manipulation", ACM, New York, 128 (1971).

Practical Fast Polynomial Multiplication

Robert T. Moenck

Dept. of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

ABSTRACT

The "fast" polynomial multiplication algorithms for dense univariate polynomials are those which are asymptotically faster than the classical $O(N^2)$ method. These "fast" algorithms suffer from a common defect that the size of the problem at which they start to be better than the classical method is quite large; so large, in fact that it is impractical to use them in an algebraic manipulation system.

A number of techniques are discussed here for improving these fast algorithms. The combination of the best of these improvements results in a Hybrid Mixed Basis FFT multiplication algorithm which has a cross-over point at degree 25 and is generally faster than a basic FFT algorithm, while retaining the desirable O(N log N) timing function of an FFT approach.

The application of these methods to multivariate polynomials is also discussed. The use is advocated of the Kronecker Trick to speed up a fast algorithm. This results in a method which has a cross-over point at degree 5 for bivariate polynomials. Both theoretical and empirical computing times are presented for all algorithms discussed.

Introduction

One of the most frequently used algorithms in any algebraic manipulation system is multiplication of expressions. For example:

$$(e^{y/2}\sin(x) - \cos(x))*(\sin(x) - e^{y/2}\cos(x)).$$

One way we might compute this product is to view it in the polynomial ring $Q[e^{y/2}\sin(x), \cos(x)]$, i.e. the functions are considered indeterminants over the ring Q. Then we might form the product using a polynomial multiplication algorithm:

$$[-(e^{y/2})^2\sin(x)\cos(x) + (\sin^2(x)+\cos^2(x))e^{y/2} - \sin(x)\cos(x)].$$

Finally, we might simplify the resulting product to:

$$e^{y/2} - 1/2(e^y+1)\sin(2x).$$

This means we can break the general problem of expression multiplication into two subproblems:

a) polynomial multiplication and
b) simplification of expressions.

Ignoring the latter completely, it is the former problem that this paper considers. Since a lot of polynomial multiplication, in one form or another, is done in an algebraic manipulation system, we would like to use the best possible algorithm. The measure of "best" we will use here is the number of steps used in the algorithm.

A second reason for interest in the problem is that over the last several years, a class of algorithms, for operations on polynomials have been developed, whose efficiency (the number of steps they use) is related to the efficiency of polynomial multiplication. In simple terms, if polynomial multiplication can be done quickly, then these algorithms can be performed quickly. A partial list of such algorithms based on an O(N log N) multiplication algorithm is given below.

Polynomial Operation	Operation Count	Reference
Multiplication	$O(N \log N)$	[Pol 71]
Division with Remainder	$O(N \log N)$	[Str 73]
GCD and Resultants	$O(N \log^2 N)$	[Moe 73]
Multipoint Evaluation	$O(N \log^2 N)$	[Bor 74]
Interpolation	$O(N \log^2 N)$	[Bor 74]
Factoring	$O(N \log^4 N)$	[Moe 75]

A list of some fast algebraic algorithms

In the case of dense univariate polynomials of degree n, the classical polynomial multiplication algorithm uses $O(n^2)$ steps to compute a product. Workers in the field of algorithmic complexity have developed several algorithms for polynomial multiplication. These are the so-called "fast" algorithms which use O(f(n)) steps to form a product. They have the property that:

$$\lim_{n\to\infty} \frac{f(n)}{n^2} \to 0$$

i.e. they are asymptotically faster than the classical method. In more concrete terms, for large degrees they are faster than the classical method and this superiority increases as n does.

The problem with these "fast" methods is that they tend to be impractical. The degree at which they start being better than the classical method (the cross-over point) is fairly high. In fact the cross-over points tend to be so high that they are larger than most problems encountered in algebraic manipulation systems. This paper sets out to answer the questions:

1) Can a fast algorithm be made practical?
2) What is the best all-round polynomial multiplication algorithm?

In section 1, we review the classical method and two fast methods and look at their performance. The performance of the algorithms is investigated both from the theoretical point of view and in terms of the run times of Algol-W implementations of the algorithms. Section 2 deals with improvements to one of the fast methods. These lead to an algorithm with the same asymptotic properties as the best of the fast algorithms, but is more practical and in general, faster than such a method. In section 3, we examine methods for multivariate multiplication, and suggest a new approach to this problem. Finally, in section 4 we summarise the results and see what conclusions can be drawn.

Three Basic Methods

In the first two sections we will consider dense univariate polynomials i.e. polynomials $a(x) \in R[x]$

where $a(x) = \sum_{i=0}^{n} a_i x^i$ and $a_i \neq 0$ for $0 \leq i \leq n$.

The base ring R will usually be a finite field Z_p. For some of the algorithms, the base ring could equally well be the integers Z or rationals Q or any other suitable number ring. Coefficients in such rings can be mapped into the finite field Z_p and recovered by means of the Chinese Remainder Algorithm (cf. [Lip71], so there is no loss of generality in considering only finite fields.

1.1) The Classical Method

The Classical Method for polynomial multiplication is the one which is commonly employed in algebraic manipulation systems. If we have two polynomials:

$$a(x) = \sum_{i=0}^{n} a_i x^i \quad , \quad b(x) = \sum_{j=0}^{m} b_j x^j$$

then their product c(x) can be written as:

$$1.1)\ldots\ c(x) = a(x)*b(x) = \sum_{\ell=0}^{m+n} \Big(\sum_{\ell=i+j} a_i * b_j \Big) x^{\ell}$$

Expression (1.1) can be directly translated into an algorithm for computing this product. This algorithm uses:

(n+1) (m+1) coefficient multiplications and

nm coefficient additions.

For a total of : $2mn + (m+n) + 1$ coefficient operations.

If n = m this is : $2n^2 + O(n)$ operations.

While this function only counts the number of arithmetic operations which are used in the algorithm, all other types of operations (e.g. subscript calculations, store, fetch etc.) will increase proportionally.

1.2) The Fast Fourier Transform Method

In recent years the study of algorithmic complexity has produced a number of multiplication algorithms having operation counts which grow more slowly than $O(n^2)$. The best of these to date is the one based on the Fast Fourier Transform (FFT) (cf. [Pol 71]). Gentleman and Sande [Gen 66] appear to have been the first to propose the use of the FFT for polynomial multiplication.

The FFT provides an efficient method for evaluating a polynomial at M roots of unity $\{w_i\}$ in a finite field. Since evaluation of a polynomial induces a homomorphism from the polynomial ring to the ring of coefficients, the values of the product polynomial can be found by multiplying the values of the multiplicants pairwise.

i.e. if $c(x) = a(x) * b(x)$
then $c(w_i) = a(w_i) * b(w_i)$.

Given the values of the product $\{c(w_i)\}$ at the points $\{w_i\}$, the coefficients of the product can be recovered by interpolation. This interpolation can be performed by means of the inverse FFT.

The advantage to this method is that the FFT can be performed very quickly. If $M = 2^r$ then this algorithm uses:

2 Forward FFTs	$= 2(M + 3/2\ M \log M)$
1 Multiplication of values	$= M$
1 inverse FFT	$= 2M + 3/2\ M \log M)$

For a total of : $9/2 \log M \log M + 5M$ coefficient operations.

Note: all logarithms used in this paper will be base 2 unless otherwise specified.

We will postpone the details of computing the FFT until section 2.1. In applying the FFT method to multiply two polynomials of degree m and n the usual approach is to choose $r = \lceil \log (m+n+1) \rceil$. Then the polynomials are considered to be of degree $M-1 = 2^r-1$ by adding a suitable number of zero leading coefficients. This approach does handicap the algorithm by making the polynomials appear larger than necessary, but does not destroy the desired O(M log M) operation count.

Applying this method to two polynomials of degree n, the number of steps in the algorithm is:

$$9N \log N + 19 N$$

where $N = 2^k$, $k = \lceil \log (n+1) \rceil$.

1.3) Comparison of the Methods

Table 1 compares the two methods both in terms of their operational counts and in terms of the actual run times of an Algol-W implementation of the algorithms. It can be seen that the operation count predicts that the FFT method will start being more efficient than the classical method when multiplying polynomials of degree 31.

The empirical run-times bear out this cross-over point. They also illustrate the characteristic quadratic behaviour of the classical method and the slightly faster than linear behavior of the FFT method. This table of performance might be used to advocate the use of the FFT method as the standard algorithm to be used in an algebraic manipulation system. However, several problems arise:

a) For degrees below the cross-over point the FFT method is very much worse than the classical method. This problem might be resolved by combining the two methods and using the FFT method for large problems and the classical for small.

Even if this combined approach is taken there are problems.

b) The cross-over point is quite high. Although products of degree 31 do appear in algebraic manipulation, they do not arise that frequently. Is it worth the trouble of implementing a fast algorithm if it will only be used occasionally?

c) The table does not illustrate the fact that the FFT is performed on sequences of length 2^k. This means that to multiply two polynomials of degree 32, a sequence of length 128 must be used. For this case the implementation of the FFT method takes 2.3 time units and the classical method about 1.1 units. In other words, the FFT method has not one, but three cross-over points with respect to the classical method (at degrees 31, 50 and 70). (see Fig. 1)

It was with these problems in mind that an attempt was made to improve on the existing multiplication algorithms.

1.4) Karatsuba's Algorithm

The first attempt involved looking at an alternative to the FFT method. It was hoped that a different algorithm might be more efficient than the classical method, but not much worse than the FFT method. The method used is due to Karatsuba (see [Kar 62]). Fateman [Fat 72] calls this algorithm SPLIT.

Karatsuba's algorithm applied to two polynomials of degree n-1:

$$a(x) = a1(x)x^{n/2} + a0(x)$$

$$b(x) = b1(x)x^{n/2} + b0(x)$$

where deg(a1), deg(a0), deg(b1), deg(b0) $\leq$ n/2-1.

Then: $c(x) = a(x) * b(x)$

$$= (x^n + x^{n/2})\, a1*b1 + (x^{n/2} + 1)$$

$$a0*b0 - x^{n/2}(a1 - a0)*(b1-b0)$$

The three products of half the degree are formed by applying the algorithms recursively. With care, the addition of the coefficients of the products can be done in 7/2n steps. Therefore, the number of steps used by the algorithm to multiply two polynomials of degree n-1 can be expressed in the following recurrence relation:

$$T(n) = 3T(n/2) + 7/2n$$

$$= 23/2\, n^{\log 3} - 7n$$

$$= 23/2\, n^{1.585} - 7n \text{ coefficient operations.}$$

This means that Karatsuba's algorithm is more efficient than the classical method, but not as efficient as the FFT method, for large inputs. However, it was hoped that the simplicity of the algorithm (in its recursive form it requires about as many line of ALGOL code as the classical method) might give it a lower cross-over point with respect to the classical method.

Table II compares the empirical run times of the classical method with Karasuba's algorithm. The second column gives the time for the first version of the algorithm, where the recursion was carried out down to the level of the coefficients (i.e. degree0). It may be seen that at degree 31 Karatsuba's algorithm takes three times longer than the classical method. Also the method exhibits the same poor performance for small degrees as the FFT method.

On reason for the poor performance of the algorithm was that it had to labour very hard to perform many operations both in the form of procedure calls and arithmetic operations close to the base of its recursion. Therefore, one way to improve the algorithm would be to stop the recursion at some small degree and perform this multiplication classically. The remainder of the columns of Table II show the times when the base of recursion was polynomials of degree 1, 3 or 7.

It can be seen that KPM-7 achieves a cross-over point at degree 15, which is the best of the fast methods. However, at about degree 63 the FFT method is almost as fast as KPM-7. For larger degrees the KPM algorithm will be slower than the FFT method. An advantage of the KPM algorithm shared by the classical method, is that it can be carried out in the original polynomial ring. This means that coefficients do not require any mapping to a residue representation with its attendant overhead in computation.

The FFT Revisited

2.1) Two Views of the FFT

A second attempt involved looking for ways to improve the FFT method. In order to discuss these,

it is first necessary to review the algorithm for computing the FFT.

There are in fact many algorithms for computing FFTs. Each has its advantages, and the choice of which one to use, depends on the nature of the problem. There is a considerable literature on the FFT, Aho et al [Aho 74] give a brief survey. Gentleman and Sande [Gen 66] discuss several such algorithms and compare their merits.

The first view considered here is the "evaluation" form. For this the transform of a polynomial a(x) of degree N is considered as evaluation of a(x) at the points $\{w^i\}$ where $0 < i < N$ and w is a primitive N-th root of unity of the finite field Z_p, i.e. $w^N = 1$, $w^j \neq 1$ for $0 < j < N$.

This can be viewed as:

$$a(w^i) = \sum_{j=0}^{N-1} a_j w^{ij}$$

$$= \sum_{j=0}^{N/2-1} a_{2j} w^{2ij} + w^i \sum_{j=0}^{N/2-1} a_{2j+1} w^{2ij}$$

$$= a_{even}(w^{2i}) + w^i \; a_{odd}(w^{2i})$$

For $0 < i < n$.

Note: $a'(w^{2(N/2+i)}) = a'(w^N w^{2i}) = a'(w^{2i})$

where a'(x) is $a_{even}(x)$ or $a_{odd}(x)$.

Now $a_{even}(x)$ and $a_{odd}(x)$ are two polynomial forms, each with half the number of terms of a(x), which are to be evaluated at a point w^{2i} which is also a root of unity. Thus the same process can be applied. The inverse transform of $\{a(w^i)\}$ can be computed as:

$$\sum_{i=0}^{N-1} a(w^i) w^{-ij} = \sum_{i=0}^{N-1} w^{-ij} \sum_{\ell=0}^{N-1} a_\ell w^{i\ell}$$

$$= \sum_{\ell=0}^{N-1} a_\ell \left(\sum_{i=0}^{N-1} w^{i(\ell-j)} \right) = N \, a_j$$

since $\sum_{i=0}^{N-1} w^{ik} = \begin{cases} N \text{ if } k = 0 \bmod N \\ 0 \text{ otherwise.} \end{cases}$

$\{w^{-1}\}$ is also a primitive root of unity and so the same algorithm as the forward FFT can be used.

This leads, naturally to a recursive algorithm for computing the transform. However, for an actual implementation, a recursive algorithm is not the most suitable since recursion is usually an expensive operation on many computers. It is possible to express this algorithm in an iterative "bottom-up" form which is much more efficient.

A second consideration is the storage used by the algorithm. It would seem that extra storage might be needed to store the $a_{even}(x)$ and $a_{odd}(x)$ results at each level, so that the coefficients of a(x) are not overwritten while they are still needed. In fact the algorithm can be coded in such a fashion that it works "in-space" and the values which are overwritten are those which are no longer required.

A peculiarity of the algorithm is the order in which it selects coefficients. Selecting even and odd parts means that the lowest order bit of the index of a coefficient is used as a selector. If this is carried through several levels, it is apparent that the indices of the coefficients are being read as binary numbers from right to left, rather than left to right as is conventional. This is the digit reverse form discussed by Cooley and Tukey [Coo 65]. One problem with this method of indexing is that it is necessary to perform "random" accessing of coefficients to apply the algorithm. This is easy if the coefficients are stored in an array, but in the realm of symbolic mathematics, polynomials are usually represented by linked lists ordered by exponent and this mode of accessing is quite unacceptable.

An alternative algorithm can be obtained by viewing the computation as being carried out modulo $x^N - w^N$.

Hence: $a(x) = \sum_{i=0}^{N-1} a_i x^i \bmod x^N - w^N$

$$= \left(\sum_{i=N/2}^{N-1} a_i x^{i-N/2} \right) x^{N/2} + \sum_{i=0}^{N/2-1} a_i x^i$$

$$= a_{high}(x) \; x^{N/2} + a_{low}(x)$$

Then $a(x) \equiv a(x) \bmod x^{N/2} - w^{N/2}$

$$= a_{low}(x) + a_{high}(x) \; w^{N/2} = a'(x)$$

and $a(x) \equiv a(x) \bmod x^{N/2} + w^{N/2}$

$$= a_{low}(x) - a_{high}(x) \; w^{N/2} = a''(x)$$

Note that $x^{N/2} + w^{N/2} = x^{N/2} - w^N$ and so the process can be recursively or iteratively applied to a'(x) and a"(x) until the values of $a(x) \bmod x - w^i$ are reached. It is apparent that a'(x) and a"(x) can be generated simultaneously by scanning the top and bottom halves of a(x) sequentially. This algorithm can also be performed "in-space". Perhaps less obvious, is the fact that, the powers of w used by the algorithm are accessed in digit reverse order and that the transformed representation of the polynomial is also permuted in this fashion.

This is less serious than the previous case since the powers of w can be generated in this order in linear time. Secondly, since the products of two transformed polynomials are formed pairwise, any permutation of the transformed

representation is irrelevant, and the inverse transform unscrambles the results. As Singleton [Sin 67] points out, this permutation must appear somewhere in an FFT algorithm. We will call this form of the FFT the sequential access method.

In the case of the second form of the FFT the inverse transform may be computed by forming a(x) out of a'(x) and a"(x) as follows:

$$a(x) = 1/2\ [a'(x) + a''(x)]x^{N/2} + \frac{W}{2}^{-N/2}\ [a'(x) - a''(x)]$$

$$= a_{high}(x)\ x^{N/2} + a_{low}(x)$$

For either algorithm it may be seen that each level of the FFT may be accomplished in 3/2N steps. Each level divides the problem into two subproblems of half the degree, each of which is to be evaluated at half the number of points. Therefore, the number of steps used by the algorithms exhibits the characteristic recurrence relation:

$$T(N) = 2T(N/2) + 3/2N$$

$$= 3/2\ N \log N + N \text{ steps.}$$

During the inverse transform the factor of 1/2 is not usually divided out at each stage but accumulated and divided out in a final step as $1/2^{\log N} = 1/N$. Thus the inverse FFT takes: $3/2N \log N + 2N$ steps.

2.2) A Hybrid Method

The basic strategy for the FFT polynomial multiplication algorithm applied to two polynomials a(x) and b(x) of degree n is as follows:

Basic Radix-2 FFT Multiplication Algorithm

1) Choose $N > 2n+1$ such that $N=2^{\lceil \log(2n+1) \rceil}$.

2) Perform the N point FFT on a(x) and b(x) to give the sequences $\{A_i\}$ and $\{B_i\}$.

3) Multiply the sequences pairwise to give a sequence $\{C_i\}$.

4) Perform the N point FFT on $\{C_i\}$ to give a polynomial c(x). This polynomial is the product of a(x) and b(x).

The problem with this approach is that it chooses a gross maximum degree N for the product. All products in the range $N/2 < 2n+1 < N$ are essentially treated the same way. One consequence of this is that for large problems within such a range the time for the multiplication is constant. For example, it takes no more time for n=240 than it does for n=130.

On the other hand, at the end of such a range there is a large jump in time from one interval to the next. Clearly it is not a great deal more dffficult to multiply polynomials of degree 32 than it is for degree 31. However, the basic FFT strategy does not reflect this fact.

One way we can improve on this situation is to notice that the FFT method is a powerful engine for multiplying polynomials. We can use it to best effect by giving it the "difficult" problems and solving the "easy" ones some other way. In polynomial multiplication the "easy" problems are the leading and trailing coefficients of the product. These involve only a few terms and operations and are best done using a classical algorithm. The "difficult" problems are the intermediate coefficients of the product. These involve most of the coefficients of the multiplicands and many operations.

To see how this might be done we can examine the inverse transform for the j-th coefficient of the product.

$$c_j = \frac{1}{N} \sum_{i=0}^{N-1} C_i w^{-ij}$$

$$= \frac{1}{N} \sum_{i=0}^{N-1} C_i w^{N/2i} w^{(-N/2-j)i}$$

$$= \frac{1}{N} \sum C_i (-1)^i w^{-(N/2+j)i}$$

and

$$c_{N/2+j} = \frac{1}{N} \sum_{i=0}^{N-1} C_i w^{-(N/2+j)i}$$

Then consider extracting the even order terms from each sum:

$$p_j = c_j + c_{N/2+j}$$

$$= \frac{2}{N} \sum_{i=0}^{N/2-1} C_{2i}\ w^{-(N/2+j)2i}$$

Note that the $\{p_j\}$ are just the result of an N/2 point inverse FFT and that

$$\left.\begin{aligned} c_j &= p_j - c_{N/2+j} \\ c_{N/2+j} &= p_j - c_j \end{aligned}\right\} \text{for } 0 < j < N/2$$

This means we could use an N/2 point FFT to compute the $\{p_j\}$ (i.e. the "difficult" terms) and and use a classical method to compute the leading and trailing terms. The two sets of terms can be combined to find the coefficients of the product.

This leads to an alternate strategy for a Hybrid Method computing a product of degree m.

The Hybrid Multiplication Algorithm

1) Choose $N = 2^{\lfloor \log m \rfloor}$.

2) Apply the FFT method to find the $\{p_j\}$.

3) Form the coefficients C_i classically for $0 \le i \le \lfloor m/2 \rfloor + N/2$ and $N/2+\lfloor m/2 \rfloor + 1 \le i \le m$. i.e. the leading and trailing terms.

4) Find the remaining coefficients of the product using the relations:

$$c_j = p_j - c_{N/2+j} \text{ , for } \lfloor m/2 \rfloor + N/2 + 1 \le j \le N/2+m$$

$$c_{N/2+j} = p_j - c_j \text{ , for } \lfloor m/2 \rfloor + N/2 + 1 \le j \le m$$

$$c_j = p_j \text{ , for the rest of the coefficients.}$$

Examining the algorithm, we see that step 2 takes: 9/2 N log N + 5N operations. Step 3 will take the same number of steps as multiplying two polynomials of degree m/2 - N/2 i.e. $\frac{1}{2}(m-N-1)^2 + (m-N-1)$ operations. There are (m-N) subtractions used in step 4. In summary we can state:

Theorem: The Hybrid multiplication algorithm takes:

2.1)... $(9/2\ N \log N + 5N) + \left(\frac{(m-N)}{2}\right)^2 + (m-N)$

operations, where $N = 2^{\lfloor \log m \rfloor}$. □

This can be compared to the Basic FFT method which takes:

2.2)... 9 N log N + 19N basic steps.

If $9/2\ N \log N + 14N > \left(\frac{(m-N)}{2}\right)^2 + (m-N)$ then

this method is faster than the basic FFT method. This inequality predicts that if: $N < m < N+120$ then the hybrid strategy is better than the basic FFT strategy. In fact, this cross-over point of 120 is borne out in empirical tests. In order not to make an $O(N^2)$ algorithm out of an O(N log N) algorithm, the hybrid method is used up to the point where 120 leading and trailing terms are computed classically. Above this point, the basic FFT method is used.

The performance of the hybrid strategy is shown in figures 1,2 and 3 and in table III. It leads to a much more smoothly growing timing function, which for large intervals, is faster than the basic FFT method. This Hybrid method crosses over with respect to the classical method at degree 25, a significant improvement over the degree 70 of the basic FFT method.

2.3) Mixed Basis FFTs

The FFT described in section 2.1 is called the radix 2 form since $N=2^k$. In fact the FFT need not be restricted to radix 2. N can be any highly composite number. For example radix 3 with $N = 3^j$ or $N = 2^k 3^j$, etc.

It would seem that another way to improve the efficiency of the FFT multiplication algorithm is to choose a value for N which more closely matches the size of the problem. If we allow mixed radix FFTs then we have a larger selection of Ns to choose from. However, the reason that the radix 2 FFT is the one most frequently used is that it is easiest to implement. This partly is due to the digit reverse permutation which is required for an iterative version of the algorithm.

The digit reverse permutation in radix 3 looks a little strange, but the permutation in a 2-3 radix FFT is stranger still. A number in the (2,2,...,2,3,...,3) number system has as its reverse form a number in the (3,...,3,2,...,2) number system (cf. Knuth [Knu 69]). This means that the permutation is not so easily computed.

An alternative is to use $N = \ell * 2^k$, where ℓ is some small prime (e.g. 3,5 or 7). This choice together with the sequential access algorithm allows the following variant of the FFT to be applied.

The Mixed Basis FFT Multiplication Algorithm

1) For a product of degree m, choose $N = 2^k \cdot \ell > m$ and a 2^k-th primitive root of unity w.

2) Perform the Sequential Access FFT algorithm, carrying out the computation modulo $x^N - w^{2^k}$. This means that the final step of the FFT yields 2^k pairs of polynomials $\{A_i(x)\}$ and $\{B_j(x)\}$ of degree ℓ-1 of the form:

$$A_i(x) \equiv a(x) \bmod x^\ell - w^i \text{ , } 0 \le i < 2^k$$

$$B_i(x) \equiv b(x) \bmod x^\ell - w^i \text{ , } 0 \le i < 2^k.$$

3) The products $C_i(x)$ and $A_i(x) * B_i(x) \bmod x^\ell - w^i$ can now be formed using the classical algorithm.

4) Finally the polynomials $C_i(x)$ have the inverse FFT applied to them, to yield the product c(x) in coefficient form.

Note that the results of the inverse FFT are divided by 2^k and not N in this case. Counting the cost for each of steps 2, 3 and 4 we have:

$$2(N + 3/2\ Nk) + (2^k * 2\ell^2) + (2N + 3/2\ Nk)$$

$$= 9/2\ Nk + N(4 + 2\ell) \text{ where } N = \ell \cdot 2^k > m.$$

We can summarise this in:

Theorem: The total number of steps for the Mixed Basis FFT Multiplication Algorithm is:

2.3)... $9/2\ Nk + N(4 + 2\ell)$, basic operations. □

An implementation of this algorith with ℓ = 1,3,5,7 was made. The performance curve of this code is shown in figures 1, 2 and 3 and in table III. This curve is much smoother than that given by either of the two previous methods. This is because the curve has many small plateaus,

where the Basic radix 2 FFT has only one. This technique is at least as fast as the Basic FFT strategy almost everywhere and in some cases faster than the Hybrid method. However, in general, the improvement is not as substantial as the Hybrid method.

The two improved methods can be combined together to give a Hybrid-Mixed-Basis FFT algorithm which is the best of the four methods (cf. figures 1, 2, 3 and table III). However, the improvements of the Hybrid and Mixed Basis algorithms do not combine together strictly additively and so the improvement of the combined method is not as large as might be expected.

This Hybrid-Mixed-Basis method can be tuned by choosing optimum values for ℓ and k and hence N in:

$$m/2 < N = \ell \cdot 2^k < m$$

for a product of degree m. How to find this algorithm is an object for further study. At the moment a "good" value is chosen by:

a) finding $N = 2^k > m/2$

b) while $m-N > 100$, increment N to a form $N = 2^k \cdot \ell$.

At the moment ℓ is always chosen as $\ell = 1,3,5$ or 7, but other choices might be better. In particular, if ℓ is a small multiple of 2, equation (2.3) predicts that the Mixed Basis method will be marginally faster than the Basic FFT method.

Multivariate Polynomials

So far we have only considered univariate polynomial multiplication. We have seen that the methods based on the FFT have a cross-over point about degree 25, with respect to classical multiplication. While problems of this size or larger do arise in algebraic manipulation, generally the problems are smaller. If we look at the multivariate case we see polynomials with similar low degrees but they tend to have a large number of coefficients. This raises the question, "Can the FFT methods be applied to the multivariate case?".

3.1) Recursive Classical Method

The standard approach to multiplication of polynomials:

$$a(x_1,\ldots,x_v), b(x_1,\ldots,x_v) \in R[x_1,\ldots,x_v]$$

is to view the polynomials as being represented recursively in:

$$a(X) \in R[x_1,\ldots,x_{v-1}][x_v]$$

i.e. as polynomials in the indeterminant x_v whose coeffients are polynomials in the remaining v-1 variables. Then the classical algorithm can be applied as before. Whenever the product of two coefficients is required, the polynomial multiplication algorithm calls itself recursively, until coefficients in the base ring R are found.

Let MCPM(n,v) be the number of steps required to multiply dense polynomials of degree n in each of v variables, using this method. Then:

$$\begin{aligned} MCPM(n,v) &= (n+1)^2 \text{ coefficient multiplications} \\ &\quad + n^2 \text{ additions to these products} \\ &= (n+1)^2\, MCPM(n,v-1) + n^2(2n+1)^{v-1} \end{aligned}$$

where MCPM(n,0)=1. The asymptotic form of this function is $MCPM(n,v) = O(n^{2v})$.

3.2 The Recursive FFT Method

One proposal for applying the FFT methods is to also apply them recursively. Viewing the FFT as a quick method to evaluate the polynomial at a set of points, we could use the following process.

1) Evaluate the $(n+1)^{v-1}$ coefficient polynomials in x_1 of a(X) and b(X) at m points. This yields m pairwise products of polynomials in v-1 variables to be formed. These products can be computed by applying this method recursively.

2) The coefficients of these products can then be interpolated using the inverse FFT to give the product of a(X) and b(X). Again the recursion stops at the pairwaise multiplication of base ring elements.

Let MFPM(n,v) be the number of steps required to multiply two polynomials of degree n in each of v variables. Then:

$$\begin{aligned} MFPM(n,v) &= 2(N+1)^{v-1}(m + 3/2m\log m) \\ &\quad + m\, MFPM(n,v-1) \\ &\quad + m^{v-1}(2m + 3/2m\log m) \end{aligned}$$

where $MFPM(n,0) = 1$ and $m = 2^r, \lceil r = \log(2n+1) \rceil$. The function is asymptotically $MFPM(n,v) = Q(v\, n^v \log n)$.

One problem with this method is that FFT methods are worthwhile only for large degree polynomials. Even with the modifications suggested in the previous section it appears intuitively that the polynomial must be of degree 25 in each variable before this method will pay dividends. This does not require many variables before the size of the problem is compounded out of any useful range.

3.3) The Kronecker Trick

A third possibility advocated here and in [Moe 76] is to map the multivariate problem into a univariate one and then apply an FFT method to the univariate case. Consider the multiplication of a(x,y) by b(x,y) where:

$$a(x.y) = (2y+1)x + (-y+2)$$

$$b(x,y) = (y+3)x + (4y-3)$$

Since the product will be of degree two in each variable we might try substituting $x = y^3$ in each of the polynomials, to obtain a'(y) and b'(y):

$$a'(y) = 2y^4 + y^3 - y + 2$$

$$b'(y) = y^4 + 3y^3 + 4y - 3$$

and a'(y)*b'(y) =

$$2y^8 + 7y^7 + 3y^6 - 7y^5 - 3y^4 + 3y^3 - 4y^2 + 11y - 6.$$

Taking the remainder of successive quotients with respect to $\{y^{3i}\}$, $2 > i > 0$, we invert the substitution to get:

$$(2y^2 + 7y + 3)x^2 + (7y^2 - 3y + 3)x +$$

$$(-4y^2 + 11y - 6).$$

This is the product of a(x,y) and b(x,y) as can be verified by carrying out the multiplication in some more conventional way. This method is known in the literature as "Kronecker's Trick". where?

For the general case the validity of the method is established by the following theorems shown in [Moe 76].

Theorem:

In the polynomial ring $R[x_1,\ldots,x_v]$, $v > 2$.

The mapping: $\phi : R[X] \longrightarrow R[x_1]$

$$\phi : x_i \longmapsto x_1^{n_i},\ 1 \le i \le v$$

where $n_v > \ldots > n_1 = 1$ is a homomorphism of rings.

□

Let $d_i(p)$ be the maximum degree of the polynomial p in variable x_i. The following theorem relates the $\{n_i\}$ of the mapping to d_i and estabilishes the validity of the inverse mapping.

Theorem:

Let ψ be the homomorphism of free R-modules defined by:

$$\psi : R[x_i] \longrightarrow R[x_1,\ldots,x_v]$$

$$\psi : x_1^k \longmapsto \begin{cases} 1 \text{ if } k = 0, \\ \psi(x_1^r) * x_i^q \text{ , otherwise} \end{cases}$$

where $n_{i+1} > k > n_i$, $k = q*n_i + r$, $0 \le r < n_i$

and $n_v > \ldots > n_1 = 1$.

Then for all $p(X) \in R[X]$, $\psi(\phi(p)) = p$

3.1... iff $\forall$ i, $1 \le i < v$,

$$\sum_{j=1}^{i} d_j(p) n_j < n_{i+1}.$$

□

Note that this result in effect shows that the set of polynomials, for which relation (3.1) holds, are isomorphic to their univariate images. Thus any polynomial ring operation on elements of this set, giving results in the set, will be preserved by the isomorphism. In this sense ϕ behaves like a ring isomorphism on the set of polynomials. Another way to view the mapping given in the theorems is:

$$\phi : x_i \longmapsto x_{i-1}^{m_i}\ , \text{ for } 1 < i \le v.$$

3.4) The Kronecker Trick Algorithm

We can apply these results in the following:

Kronecker Trick Algorithm

1) For two polynomials a(X) and b(X), find a bound on the maximum degree in each variable of their product:

$$m_i = d_i(a) + d_i(b) + 1.$$

2) Apply the Kronecker Trick mapping:

$$\phi : x_i \longmapsto x_{i-1}^{m_i},\ \text{for } 1=v,v-1,\ldots,2$$

to a(X) and b(X) to give two univariage polynomials $a'(x_1)$ and $b'(x_i)$.

3) Multiply the polynomials $a'(x_1)$ and $b'(x_1)$ to give a product of $c'(x_1)$. Any fast multiplication algorithm may be used.

4) Invert the Kronecker Trick mapping on $c'(x_1)$ to give a multivariate product c(X).

□

From the theorems given above, the result c(X) will be the product of a(X) and b(X). The Kronecker Trick mapping and its inverse can be accomplished by a linear scan over the terms of the polynomial. This means we can count the steps used in the algorithm as follows. Let MKPM(n,v) be the number of steps required to multiply two polynomials of degree n in each of v variables, using the Kronecker Trick and a univariate FFT method. Then:

$$MKPM(n,v) = (9/2m \log m + 5m)$$

where $m = 2^r$ and $r = \lceil \log (2n+1)^v \rceil$. Again the asymptotic form of the function is $MKPM(n,v) = O(v\, n^v \log n)$. However, the constants of proportionality are smaller in this case than in the recursive FFT algorithm

The three functions MCPM, MFPM and MKPM can be evaluated using their recursive definitions and an estimate of the cross-over points can be made. Some sample values for small n and v are displayed in table IV. The table illustrates the characteristic jumps in the timing function of the FFT methods. One effect of the Kronecker Trick is to smooth out these jumps to give a more uniform function. While the recursive FFT method is better than the Kronecker method at a few points,

generally the latter is faster. The cross-over points for the Kronecker method with respect to the classical method are shown in the following table.

variables	theoretical degree	empirical degree
2	10	6
3	7	3
4	4	
5	4	
6	3	

Some empirical tests using the Hybrid-Mixed-Basis FFT algorithm together with the Kronecker Trick were carried out. The results of these are shown in figure 4 and table IV. It can be seen that for bivariate polynomials a cross-over point at degree 5 and for trivariate polynomials a cross-over point at degree 3 is obtained.

Summary and Conclusions

We have surveyed some of the existing fast methods for computing the product of two polynomials. When contrasted with the classical algorithm for computing products, these methods tend to suffer from the common problem that they do not perform well for small degrees. A number of techniques were proposed to improve these methods. These techniques include:

1) Performing the basis step of Karatsuba's algorithm classically. If the degree of the basis was increased to 7, the algorithm attains a cross-over point at degree 15, with respect to classical multiplication.

2) Another technique used was the Hybrid FFT algorithm which computed the leading and trailing coefficients of the product classically, and left the intermediate terms to an FFT algorithm.

3) A Mixed-Basis FFT algorithm reduced the computation of one large product to that of a large number of small products, which could be computed classically.

4) A Hybrid-Mixed-Basis FFT algorithm combined the previous two methods. This last method was the best of the FFT methods considered here, attaining a cross-over point at degree 25 and having a much smoother performance curve.

The problem of multivariate multiplication was also considered. The Kronecker Trick was proposed as an alternative to the standard recursive approach to multivariate problems. Evidence was presented showing that the Kronecker Trick is superior to the standard methods. A cross-over point at degree 5 for bivariate polynomials was obtained in empirical tests.

When this research was begun, it was hoped that a definitive answer could be found for the question, "What is the best all-round polynomial multiplication algorithm(at least in the dense case)?". Instead of a definitive answer, more questions were found. Some of these were:

1) The techniques used to speed up the basic algorithms suggest that there are classes of methods for polynomial multiplication that are yet to be explored, e.g.

   a) The Hybrid method uses the classical algorithm to compute up to 60 of the leading and trailing terms of a product. Could some other algorithm (e.g. Karatsuba's) do this computation more efficiently?

   b) The Hybrid method uses the FFT to compute half of the terms of the product. Could the FFT be used to compute a quarter of the terms, leaving the rest to be formed classically? This might reduce the cross-over point for these methods.

   c) What is the optimum choice for the division of labour in a Hybrid-Mixed-Basis FFT method?

2) The FFTs used in the empirical tests were the standard finite field methods (cf. [Pol 71]). Recently, Mersenne and Fermat Transforms which replace multiplication by powers of w with clever shifting, have been proposed. Agrarwal and Burrus [Agr 74] claim that such methods are faster than FFT method by a factor of 3. The techniques used in the Hybrid and Mixed Basis methods are not senstitive to the kind of transform employed, and so could be used in conjunction with such methods. Would this yield a better algorithm?

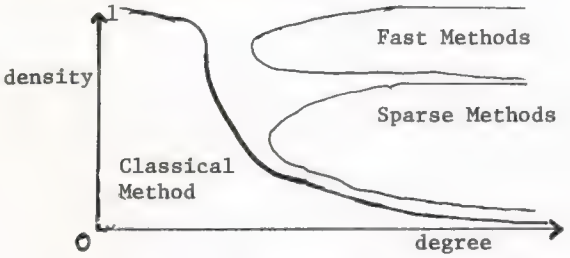
3) The products considered in this paper have all been balanced in the following sense: for a product of polynomials of degrees n and m: $n < m < 2n$. If they are unbalanced i.e. $n \ll m$, then usually one segments the polynomial of degree m into a number of "chunks" each of approximate degree n, and forms the product of these with the polynomials of degree n. What is the best way to do this?

4) Last, but not least, comes the question of sparse polynomials. These have been completely ignored in the present work because the dense case has enough problems of its own. The classical algorithm can perform as $O(t^3)$ when multiplying polynomials with t terms. Johnson [Joh 74] and Horowitz [Hor 74] have proposed $O(t^2 \log t)$ algorithms and Gustavson and Yun have an $O(t^2)$ algorithm for performing unordered sparse polynomial multiplication.

Current opinion holds that fast techniques are hopeless for sparse polynomials. Is this in fact true? The Kronecker Trick maps a dense multivariate problem onto a sparse univariate one. For example the bivariate case where half the terms are zero. In a sense, the classical algorithm knows that the problem is sparse, and yet it still looses to the FFT techniques.

One might define the density of a polynomial

as the fraction of non-zero coefficients, when the polynomial is viewed in its dense form. The best multiplication algorithm to use, is a function of the density and the degree of the result. The optimum regions for the various methods are probably as shown below.



At the moment a best all-round algorithm for polynomial multiplication is probably a poly-algorithm which would examine the density and degree of the result and choose an appropriate method to compute the product. In order to generate such a poly-algorithm, the boundaries between these optimum regions must be known. Exactly where the boundaries between the methods lie is another question for further research.

Acknowledgement

The author would like to thank Professor J. Lipson for introducing him to FFT techniques.

References

Agr 74 Agarwal R.C. and Burrus C.S.: Number Theoretic Transforms to Implement Fast Digital Convolution; IBM Research Report RC5025, 40 pp, (1974).

Aho 74 Aho A., Hopcroft, J., Ullman J.: The Design and Analysis of Computer Algorithms; Addison-Wesley, Reading, Mass (1974).

Bor 74 Borodin A. and Moenck R.: Fast Modular Transforms; J. Computer and System Sciences, Vol 18, pp 366-387 (1974).

Coo 65 Cooley J.W. and Tukey J.W.: An Algorithm for Machine Calculation of Complex Fourier Series; Math. Comp., Vol 19, pp 297-301 (1965).

Fat 74 Fateman R.J.: Polynomial Multiplication, Powers and Asymptotic Analysis: Some Comments; SIAM J. of Computing, Vol 3, pp 196-213 (1974).

Gen 66 Gentleman W.M. and Sande G.: Fast Fourier Transforms - For Fun and Profit; Proc AFIPS 1966 FJCC, Vol 29, pp 563-578.

Gus 76 Gustavson F. and Yun D.: Arithmetic Complexity of Unordered Sparse Polynomials; preprint.

Hal 71 Hall A.D.: The Altran System for Rational Function Manipulation - A Survey; CACM, Vol 14, pp 517-521 (1971)

Hor 75 Horowitz E.: A Sorting Algorithm for Polynomial Multiplication; JACM, Vol 22, No. 4, pp 450-462 (Oct 1975).

Joh 74 Johnson S.C.: Sparse Polynomial Arithmetic; Proc EuroSam 74 Conf, pp 63-71 (1974).

Kanada E. and Goto E.: A Hashing Method for Fast Polynomial Multiplication.

Kar 62 Karatsuba A.: Doklady Akademia Nauk SSSR, Vol 145, pp 293-294 (1962).

Knu 69 Knuth D.: Seminumerical Algorithms; Addison-Wesley, Reading, Mass (1962).

Lip 71 Lipson J.D.: Chinese Remainder and Interpolation-Algorithms; In: Petrick S.R. (ed.) Proc. 2nd Symp. on Symbolic and Algebraic Manipulation, New York, ACM, pp 188-194 (1971).

Moe 73 Moenck R.: Studies in Fast Algebraic Algorithms; University of Toronto, Ph.D. Thesis, Sept 1973.

Moe 74 Moenck R.: On the Efficiency of Algorithm for Polynomial Factoring; to appear in Math. Comp.

Moe 75 Moenck R.: Another Polynomial Homomorphism; to appear in Acta Informatica.

Pol 71 Pollard J.M.: The Fast Fourier Transform in a Finite Field; Math. Comp., Vol 25, No. 114, pp 365-374 (April 1971).

Sin 67 Singleton R.G.: On Computing the Fast Fourier Transform; CACM, Vol 10, No. 10, pp 647-654 (Oct 1967).

Str 73 Strassen V.: Die Berechnungs Komplexitaete elementar symetrischen Funktionen und von interpolations Koeffizienten; Numerische Math., Vol. 20, pp 238-251 (1973).

	Classical Method		FFT Method	
N Deg + 1	2N - 2N + 1	Empirical Secs./60	Theoretical 9N log N + 19N	Empirical Secs./60
2		0.01		0.12
4		0.03		0.16
8		0.08		0.3
16	481	0.29	880	0.55
32	1985	1.10	2048	1.05
64	8055	4.25	4672	2.3
128		17		5.0
256		66		11

Table 1

A comparison of the classical vs. the FFT method of univariate polynomial multiplication. Times are taken for an Algol-W code running on an IBM 360-75 for multiplication of polynomials with coefficients in the finite field GF(40961). Note that both the theoretical and empirical times indicate a cross-over point at degree 31.

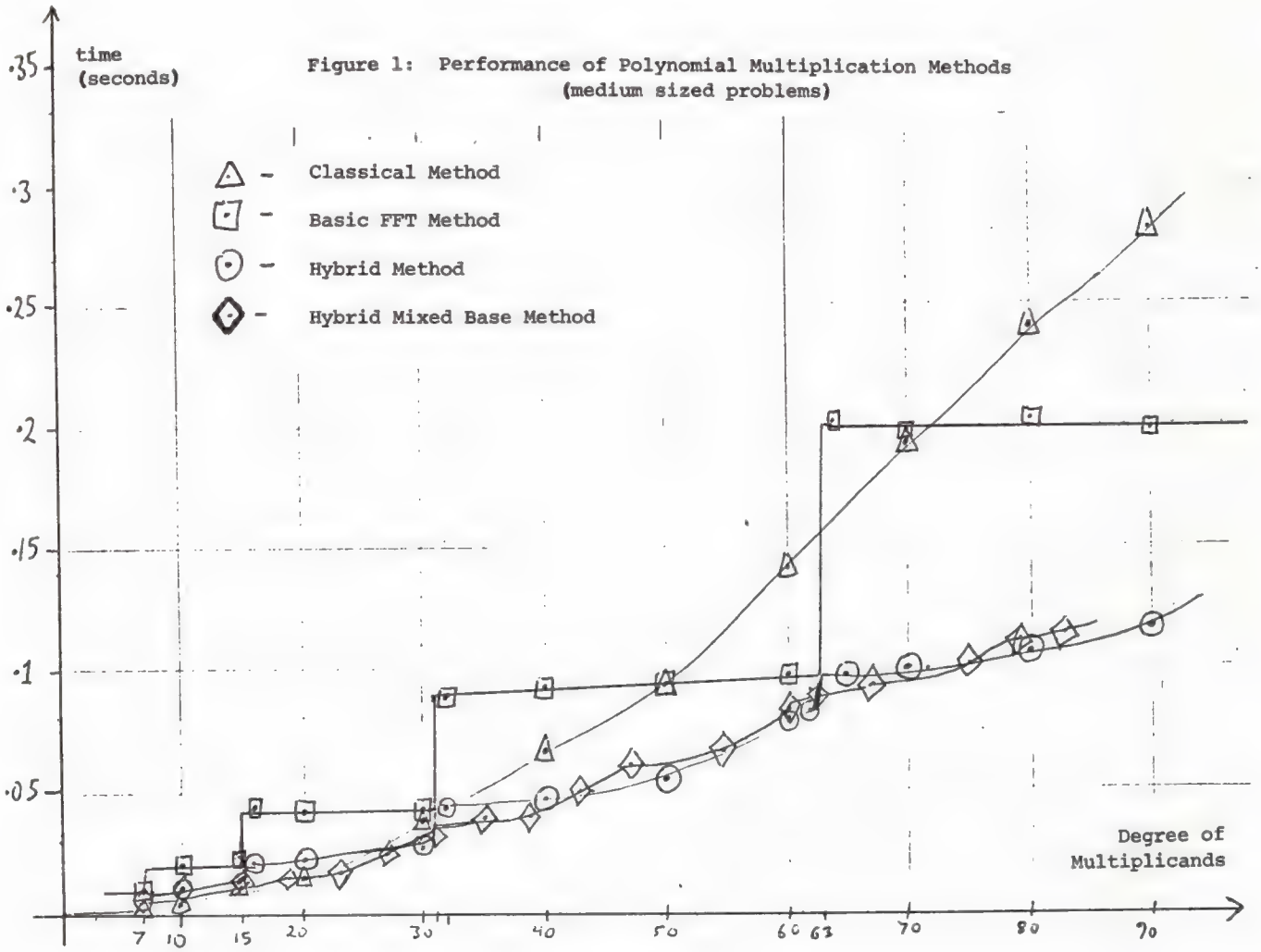
Deg	Classical	KPM-0	KPM-1	KPM-3	KPM-7	FFT
3	0.7	3	1.1	0.8		-
7	2	8	4	3	2	8
15	8	30	15	10	8	20
31	38	91	50	30	27	42
63	113	-	150	100	90	97

Table II

A table comparing empirical times in milli-seconds for classical multiplication with a class of Karatsuba algorithms. The times were obtained from an Algol-W code running on an IBM 360-75. KPM-n indicates the basis of degree n multiplication was performed classically.

The times for an FFT algorithm operating on polynomials of the same degree are included for contrast. Note that the classical method and Karatsuba's algorithm were carried out over the integers. This accounts for the discrepancy with Table I. The FFT times are for computation in a finite field. Each finite field operation involves an extra division to compute the residue of the result of an integer operation.

Figure 1: Performance of Polynomial Multiplication Methods
(medium sized problems)



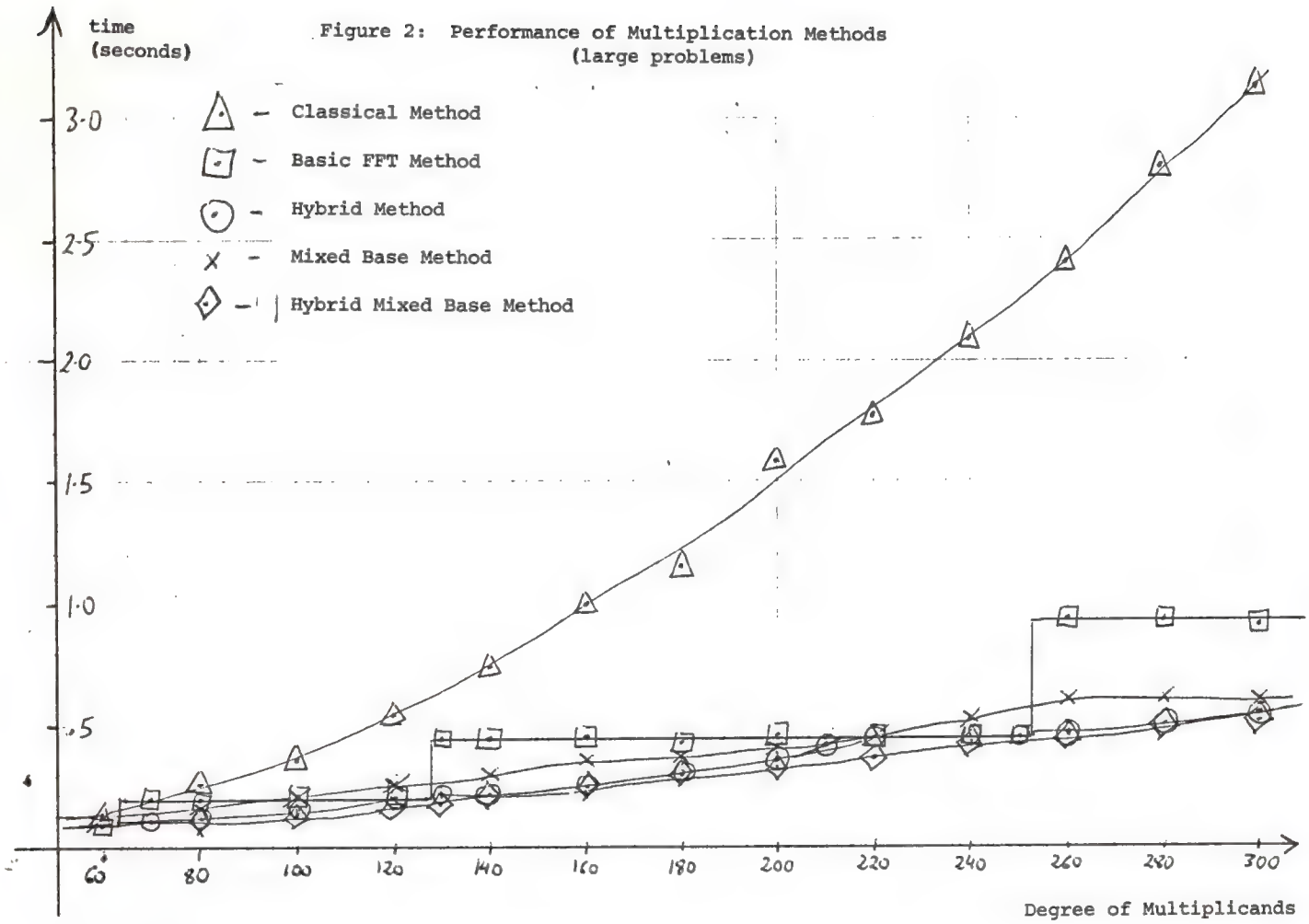


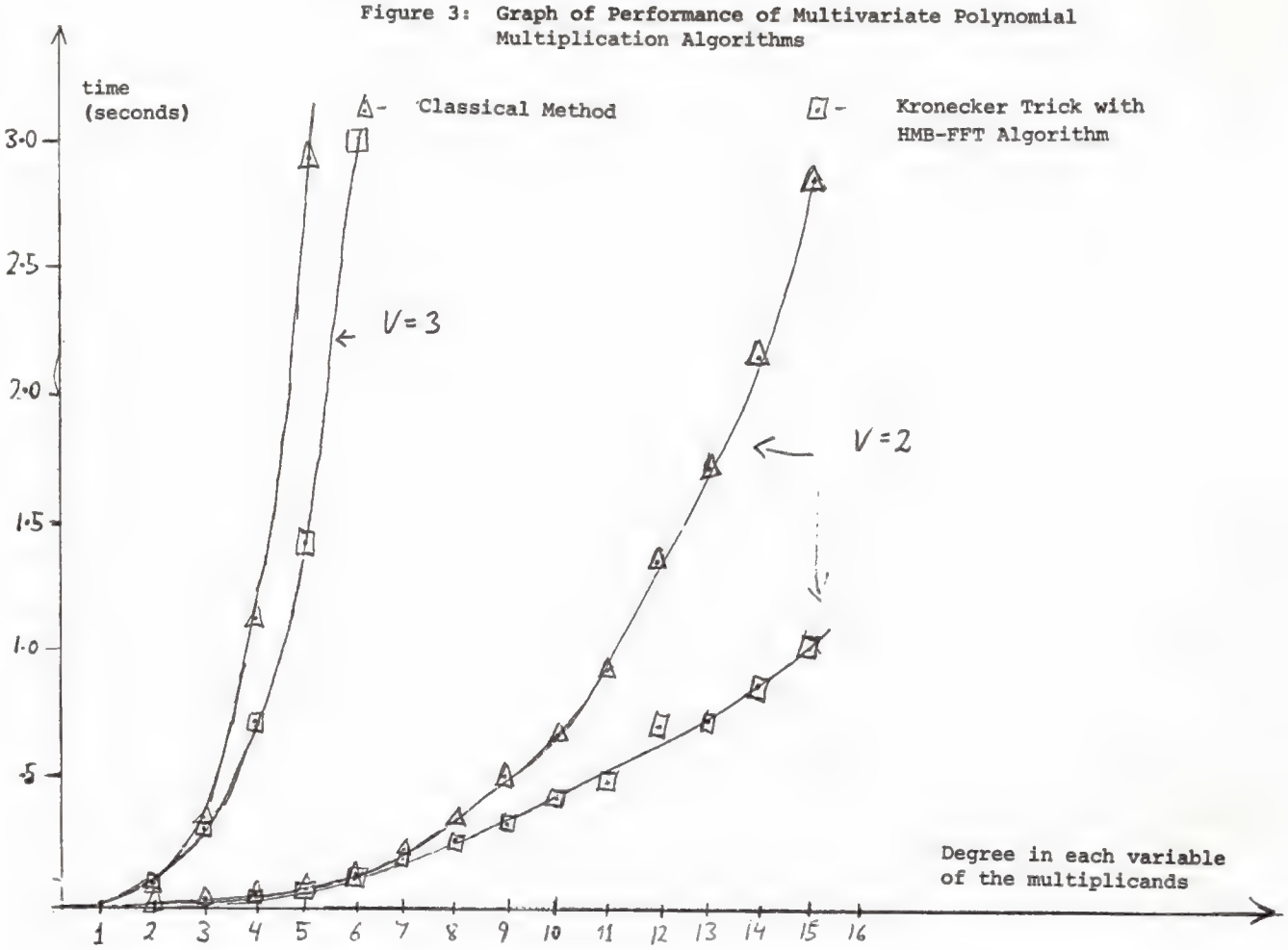
Figure 2: Performance of Multiplication Methods
(large problems)

N	Classical	Basic FFT	Hybrid	Mixed Basis	HMB-FFT
10	5.5	22	15	16	11
20	16.6	44	28.3	33	22
30	33	44	35	55	33
40	66	94	62	72	50
50	94	88	68	88	61
60	139	94	88	111	83
70	194	205	117	133	94
80	250	200	125	166	111
90	294	206	137	166	139
100	377	206	163	200	150
150	866	450	255	294	250
200	1600	455	412	444	350
250	2238	433	530	511	472
300	3128	906	583	606	572

Table III

This table contrasts the run-times for the Classical, Basic FFT, Hybrid, Mixed Basis and Hybrid Mixed Basis univariate polynomial multiplication algorithms. The times (in milli-seconds) are taken from an Algol-W code running on an IBM 360-75.

Figure 3: Graph of Performance of Multivariate Polynomial Multiplication Algorithms



var v	de N	Classical steps	time	Kronecker steps	time	FFT steps
2	2	137	8.33	880	13.3	1192
	4	1169	43.3	4670	50	7264
	6	4633	135	10496	130	7712
	8	12833	345	23296	257	47392
	10	28841	696	23296	405	48480
	12	56497	1333	51200	650	49568
	14	100409	2200	51200	850	50656
	16	165953		111616		328960
	18	259273		111616		331520
	20	387281		111616		334080
3	2	1333	80	4672	85	10328
	4	30521	1166	51200	727	121824
	6	233101	6780	241664	3012	134368
	8	1057969		520192		1560608
	10	35333861		11114112		1617184
4	2	12497		51200		85000
	4	774689		520192		1977184
	6	11501041		2375680		2226720
5	2	114937		241664		687128
	4	19472201		5046272		31774944
6	2	1047257		1114112		5518408
	4	487749809		9961472		509099104

Table IV

A table comparing the number of steps and the run times (in milli-seconds) of the Recursive Classical, Kronecker Trick and Recursive FFT algorithms for multivariate polynomial multiplication.

Arithmetic Complexity of Unordered Sparse Polynomials

F. Gustavson
D. Y. Y. Yun

IBM Thomas J. Watson Research Center
Yorktown Heights, New York

1. Introduction

Polynomial arithmetic has received considerable attention in recent years. In terms of algorithm analysis and computational complexity, Knuth [1] and Aho, Hopcroft, and Ullman [2] provide solid foundations for the initiation of any newcomer to this area. However, it was S. C. Johnson [3] who first reported his extensive work in the study of arithmetic operations on sparse polynomials in 1974. He was able to establish that the multiplication of two sparse polynomials, with n and m non-zero terms, can be done in time mn log n. In terms of actual implementation of polynomial representation and arithmetic, it is fair to say that almost all symbolic and algebraic manipulation systems have a sparse canonical representation and related arithmetic operations thereof. The word "canonical" generally implies some kind of ordering on the exponents of the polynomials, and canonical representations usually make the equivalence recognition, in particular the zero recognition, problem easier. However, testing for equality of two polynomials is equivalent to recognizing whether their difference is zero. In view of our result on subtraction of unordered polynomials, we realize that ordering plays a non-essential role. Another important usage of ordered representation is on output. Human users of symbolic systems demand orderly displays of polynomials. But, again this is only a human preference rather than an inherent requirement for ordered representations. Thus, why not consider using unordered representations of sparse polynomials, especially if there are computational efficiencies to be gained.

In this paper, we show that addition and subtraction of unordered sparse polynomials can be done in time m+n and multiplication in time mn. So we see already that if a particular sequence of polynomials computation involves a combination of these arithmetic operations as well as tests for equivalence or zero while intermediate expressions are not important enough to be displayed, then the unordered representation could amount to substantial savings in computation time. The visual demand of human users can still be satisfied at the cost of a single sort of the final answer as opposed to sorting each intermediate result in the case of ordered representations.

2. Unordered Sparse Polynomial Addition and Multiplication

Suppose we are given two sets S and T of size m and n, respectively, of unordered positive integers.

Let each element of S and T be no greater than N and suppose we have a temporary vector of size N initialized to zero. The concept of an unordered merge allows one to find the elements of R=S∪T in O(m+n) operations; unordered merge means to remove or combine the equal terms of the unordered operands. Unordered merging uses both a full (X) vector and a sparse (R) vector to get its result. First we place the elements of S into R and X; i.e., R←S and X(s)←1 when s∈S. Next for each element t∈T, we test whether X(t) is 0 and add t to R and set X(t)=1 when it is; otherwise nothing is done. Finally, we use R to rezero X; i.e., for each r∈R we set X(r)=0 which leaves X in its initialized state.

Suppose we are given two univariate polynomials $f = \sum_{i=1}^{n} a_i x^{\alpha_i}$ and $g = \sum_{j=1}^{m} b_j x^{\beta_j}$ where the α_i and β_j are unordered. We assume that exponents are distinct and the coefficients are nonzero. The zero polynomial is represented as a polynomial with no terms. We show that the sum and product f and g can be computed in time O(m+n) and O(mn) operations respectively. The reason for considering sum is that multiplication produces an unordered result even if f and g are ordered; hence, it becomes important to perform addition and subtraction on unordered operands. In what follows we assume that our polynomials will never have exponents greater than some upper limit L and that we have a full vector X of size L initially set to zero. In the multivariate case we need space for d-vectors of size L_i for each variable where d is the maximum number of variables.* Thus, $L_1+L_2+\ldots+L_d$ is the total space requirement.

*If one maps $\alpha_i = (\alpha_{i1},\ldots,\alpha_{id})$ into an integer via $\alpha_1=(\ldots(\alpha_{i1}\ell_2+\alpha_{i2})\ell_3+\ldots)\ell_d+\alpha_d$ then space for $\prod_{\nu=1}^{d}\ell_\nu$ is required. This will usually be too large a space penalty; a hash scheme is then recommended to reduce the space requirement for X.

To compute $\sum_{i=1}^{n} a_i x^{\alpha_i} + \sum_{j=1}^{m} b_j x^{\beta_j} = \sum_{k=1}^{\ell} c_k x^{\gamma_k} = s(x)$ we assume that n≤m; if not, reverse the role of f and g. In the ADD algorithm to follow we first copy f in X; the sparse copy of X remains in f. Next we compare g with f for exponent equality. When equality occurs we add the g and f term together in X; otherwise we copy the g term into s(x), the sum. Note that this term cannot cancel. Finally we pass over f again to look for cancellation, set X back to zero, and to put the rest of the sum, s(x), away. The addition algorithm follows:

Addition Algorithm

```
k←0; i←1; j←1;
   WHILE (i≤n) {X(α_i)←a_i; i←i+1}
   WHILE (j≤m)
      {IF (X(β_j)=0) THEN
       {k←k+1; c_k←b_j; γ_k←β_j}
       ELSE
       {X(β_j)←X(β_j)+b_j}
      j←j+1}
i←1;
   WHILE (i≤n)
      {IF (X(α_i) ≠ 0)
      THEN {k←k+1
         c_k←X(α_i)
         γ_k←α_i
         X(α_i)←0}
      i←i+1}
ℓ←k.
```

Multiplication is equally simple. Again assume m≤n. Besides X we use a vector XS (a sparse copy of X) and a scalar e as temporary storage.* First we check to see if m is one; if it is the result is trivial and we transfer it from g(x) to $p(x) = \sum_{k=1}^{\ell} c_k x^{\gamma_k} = f(x)g(x)$ without using X or XS.

*The vector XS is not needed. We could use the result exponent array γ instead of XS. Later on, however, we must delete cancellations from γ and this complicates a clear description of the algorithm.

When m>1 we form mn products and place the $(i,j)^{th}$ one, $a_i b_j x^{\alpha_i+\beta_j}$, in X location XS(k) = e = $\alpha_i+\beta_j$. If X(e) = 0 when this term is formed we add e to the sparse exponent list XS and set X(e) = $a_i b_j$, otherwise we merely update X(e)←X(e) + $a_i b_j$. This procedure clearly costs O(mn). Next we reset X to zero and remove zero elements (due to cancellation) from the set of sparse unordered exponents XS. We do this by examining X(XS(i)) for i=1,...,k where k≤mn is the size of the sparse exponent set. The condition X(e) = 0 where e = XS(i) signifies cancellation; we do nothing in this event. If X(e) ≠ 0 we have a non-zero term in the final product p(x); if this event occurs we copy it there and then set X(e) = 0. This last part is O(k); hence the whole algorithm is O(mn). The multiplication algorithm follows:

Multiplication Algorithm

```
IF (m≠0) RETURN THE ZERO POLYNOMIAL
k←1; i←1; j←2;
IF (m=1) THEN
   {WHILE (i≤n){
      c_k←a_i b_1; γ_k←α_i+β_1;
      i←i+1; k←k+1}
   RETURN}
ELSE WHILE (i≤n){
   XS(k)←α_i+β_1; X(XS(k))←a_i b_1;
   k←k+1; i←i+1}
WHILE (j≤m){
   i←1;
   WHILE (i≤n){
      e←α_i+β_j
      IF (X(e)≠0 THEN X(e)←X(e)+a_i*b_j;
      ELSE {XS(k)=e; X(e)←a_i*b_j; k←k+1}
      i←i+1}
   j←j+1}
i←1; j←k; k←1;
   WHILE (i≤j){
   e←XS(i)
   IF (X(e)≠0) THEN
      {γ_k←e; c_k←X(e);
      k←k+1; X(e)←0}
   i←i+1}
ℓ←k;
```

The result on multiplication also follows from a recent result [4] on multiplying sparse matrices. Let A be a p×q matrix and B be a q×r matrix both stored in sparse row-wise format.* Then C=AB can be found in O(p,r,NA, η) operations where NA is the number of non-trivial multiplications. Let A be the 1 by N+1 matrix representing the sparse polynomial f. A has n non-zero terms. Let B be the N+1 by N+M+1 matrix associated with polynomial g; i.e.,

$$B = \begin{pmatrix} b_0 & & & b_M & \\ & b_0 & & & b_M \\ & & \ddots & & \ddots \\ & & & b_0 & & b_M \end{pmatrix}$$

The product AB is the product p(x). We have p=1, r=M+N+1, NA=N+1, and η=mn. Hence we can compute p(x) in O(mn,N+M). The N+M is required to obtain p(x) sorted and to initially zero out X whose size is N+M+1.

3. The Ordered Case

We can use the concept of unordered merge to reduce the ordering cost for multiplication from O(mn log n) to the minimum of O(ℓ log n) and O(M+N) where ℓ is the number of terms in the product and M and N are the degrees of f and g.**

---

* A sparse row-wise representation of a matrix A is given by three one-dimensional arrays. Two arrays list the column indices of the non-zero matrix entries and their numerical values. The arrangement of these elements is row-wise. The third array is a set of address pointers and the ith element of this array is the address of the column index and numerical value of the first non-zero element of the ith row of A.

** Actually M+N can be lowered to M+N-($\alpha_1+\beta_1$) where α_1 and β_1 are the terms of lowest degree in f and g.

The reason for this can be seen by studying the "exponent matrix"

$$E = \begin{pmatrix} \alpha_1+\beta_1 & \cdots & \alpha_1+\beta_m \\ \alpha_2+\beta_1 & \cdots & \alpha_2+\beta_m \\ \vdots & & \vdots \\ \alpha_n+\beta_1 & \cdots & \alpha_n+\beta_m \end{pmatrix}$$

If the exponents α and β are ordered to begin with then the rows of E are also ordered. The unordered merge process which is performed during the mn multiplying steps has the property that it marches over the rows of E and strikes out duplicates; i.e., after the unordered merge has been carried out the elements of row i of E are disjoint from the union of the first i-1 rows of E and they are still ordered. Hence the ith row of E is transformed into ordered list of exponents from row i of E, each of length $i \le m_i \le m$ where $\sum_{\nu=1}^{n} m_\nu = \ell$. It is well known that these n disjoint lists can be merged in $O(\ell \log n)$ operations. It may turn out that $M+N-(\alpha_1+\beta_1)$ is less than $\ell \log n$. In this event we can check locations X(i) for $i=\alpha_1+\beta_1 \ldots, M+N$ in a sequential manner to determine the order of exponents in p(x); this procedure is akin to an address sort.

4. The Division Problem

Division appears to be a unique arithemetic operation in that ordering seems to have an inherent importance if the operation is to get carried out efficiently. In considering the relationships of different operations with respect to the arithmetic complexities of dense (at least not sparse) polynomials, Aho, Hopcroft, and Ullman [2] showed that the complexity for multiplication and division are related by constant factors, i.e., cost (M) $\le C_1$ cost (D) and cost (D) $\le C_2$ cost (M) where C_1 and C_2 are rational constants. In the case of sparse polynomials, we observe that such a relationship may not and need not exist. For example, the quotient of two sparse polynomials, may be very dense as is illustrated by $(x^n-1)/(x-1) = x^{n-1}+\ldots+x+1$. The proof for cost (D) $\le C_2$ cost (M) relies on the formula $A/B = A*(1/B)$ and $(1/B)$ can be quite large and dense. When the issue of order is brought in, as we did with the other arithmetic operations, we are in an even more complicated situation. The question is whether the inputs to any division algorithm must be ordered or whether at least the divisor has to be ordered. Its answer will determine whether division of sparse polynomials can be made more efficient by ignoring the ordering of inputs and outputs. We have not been able to improve Johnson's division algorithm on ordered polynomials. We make the following conjecture:

> Let A and B be two unordered sparse polynomials with m and n terms respectively. Let C = AB also be unordered, so that it is computed at a cost of O(mn). Then the problem of computing any quotient B' = C/A such that B' is equivalent to B cannot be solved in O(mn) operations by any algorithm.

We shall now give our view on the essential difficulties of ordered sparse multiplication and division. The reason multiplication is less difficult than division is that the arithmetic stages can be separated from the merging stages. For multiplication we can form the exponent matrix $E = (\alpha_i+\beta_j)$ first and then merge the rows (or columns) of E, hoping to achieve the merging in

comparable time, mn, to the arithmetic stage. The essential difficulty here is the merging of E. The division problem is different and difficult in the sense that division and merging processing appears to be inseparable. Thus a good division algorithm must perform the multiplication and subtraction steps and the unscrambling (or reordering) "online".

5. Definitions of Sparsity

The computing cost of ordered sparse polynomial multiplication obtained from the sparse matrix multiplication bound is $O(M+N) + O(mn)$. This cost leads to a very interesting and useful "definition" of sparsity or density. Such a parametric definition has been lacking in the literature partly because sparsity is a loosely used and defined word, partly because of the lack of good algorithms to deal with sparse polynomials and partly due to the inherent difficulty in dealing with such entities that lacks structures.

Let us consider the problem of multiplying two polynomials f and g of equal degree and equal number of non-zero terms so $M = N$ and $m = n$. Since the cost is $2N+n^2$, the crossover point where one part of the sum becomes dominant over the other is when $n^2 = 2N$ or $(n/N) = (2/n)$. Notice that the quantity n/N is precisely the density or sparsity of the polynomials f and g. Thus using a definition of density or sparsity 2/n has the meaning that if the density of a polynomial of degree N with n non-zero terms is greater than 2/n, then the cost of multiplying is simply n^2 which corresponds to the cost of multiplying nearly dense polynomials. Otherwise the cost of multiplying is dominated by 2N, and that corresponds correctly to the situation of multiplying quite sparse polynomials. In general, when we use this definition of sparsity for f and g, 2/n and 2/m respectively, then the cost for multiplication is mn if the polynomials are denser than 2/n and 2/m and M+N if they are sparser than that.*

References

[1] Knuth, D. E., "The Art of Computer Programming. Seminumerical Algorithm," Vol. 2, Addison Wesley Publ. Co., Reading, Mass. 1969.

[2] Aho, Hopcroft, Ullman, "The Design and Analysis of Computer Algorithm," Addison Wesley Publ. Co., Reading, Mass. 1974.

[3] Johnson, S. C., "Sparse Polynomial Arithmetic," Proceedings of Eurosam '74, pp. 63-71, (ACM SIGSAM Bull., Vol. 8, No. 3, August 1974, Issue No. 31).

[4] Gustavson, F. G., "An Efficient Algorithm to Perform Sparse Matrix Multiplication," to appear.

* This remark is not strictly true. Instead of M+N we can use p log n whenever it is smaller.

Hashing Lemmas on Time Complexities with Applications to Formula Manipulation

EIICHI GOTO*,** AND YASUMASA KANADA*

* DEPARTMENT OF INFORMATION SCIENCE, UNIVERSITY OF TOKYO, TOKYO, 113, JAPAN
** INSTITUTE FOR PHYSICAL AND CHEMICAL RESEARCH, WAKOSHI, SAITAMA, 351, JAPAN

I. Introduction and Summary

Johnson[1] and Horowitz[2] applied sorting to improve time complexity of multiplication of univariate polynomials. Their results may be regarded as applications of the following LEMMA:

Sorting LEMMA. The time complexity of sorting of N items is $O(N\log_2 N)$ and that of binary search of sorted N items is $O(\log_2 N)$.

In this paper, time complexities of operation on "sets" and "ordered n-tuples" based on a hashing table search technique are presented as "Hashing LEMMAs" and are applied to formula manipulation. Unique normal forms for multivariate symbolic formulas resulting in *O(1)* time complexity for identity checks are presented. The logarithmic factor $\log_2 N$, characteristic to sorting algorithms, is shown to all disappear from time complexities of polynomial manipulations. Actual implementation of the hashing technique is outlined and actual timing data are presented in the appendix.

II. Hashing LEMMAs on Sets and n-Tuples.

(2.0) Denotations and Conventions:
In case x represents a set or an n-tuple, |x| means the number of elements.
Sets are denoted by underscored capital letter(s). Specially,
INT is the set of (all) integers;
INT0 = INT - {0}, i.e., integers except 0;
INT+ is the set of positive integers.
A BNF metaobject is denoted by embracketing a set in the underscoring notation between "<" and ">", with optional commentary un-underscored letters. This convention enables us to use both BNF and set notations. E.g., BIT = {0,1} and <Binary digIT> ::= 0|1 ₀ are equivalent definitions, where "₀" means the end of a BNF definition.
In order to present algorithms precisely and concisely, Lisp with three additional data types <ordered n-TUPle>, <SET> and <ASSociator> are used in this paper. <INTeger>, <SYMbol, i.e., nonnumeric atoms> and <CONS, i.e., data created by Lisp functions "cons" or "list"> are the three data types of ordinary Lisps. (Floating point numbers and arrays are omitted because of irrelevance to this paper.) Since the time complexity of high precision arithmetic is not the theme of this paper, the time complexities of arithmetic operations on <INT>'s are assumed to be *O(1)* for the sake of simplicity.
<IDentifiables> are defined as:
ID = INT ∪ SYM ∪ TUP ∪ SET ∪ ASS; (<CONS> ∉ ID).

While <ASS>'s are denoted as <ASS>::= (.<ID>)₀, <TUP>'s and <SET>'s are denoted in accordance with ordinary mathematical notations:
<TUP> ::= (<ID>,₀₀₀)₀; <SET> ::= {<ID>,₀₀₀}₀,
where ",₀₀₀" means nonzero repetition of the same metaobject. Specially the 0-tuple () and the null set {} are regarded equivalent to NIL, i.e.,
() ≡ {} ≡ NIL.
<CONS> is printed as cons[A;()] = (ØAØ) with extra blanks (Ø's) at both ends to discriminate them from a <TUP> printed as (A).

(2.1) A function "tcons" appends an <ID> to a <TUP>, e.g.,
tcons[A;()]=(A), tcons[{A,B};(C)]=({A,B},C).
Lisp functions "car", "cdr", "cadr" etc. work on <TUP>'s as on <Lisp LIST>'s, e.g.,
car[(A,B)]=A, cdr[(A,B)]=(B), cadr[(A,B)]=B.
<TUP>'s are uniquely represented in the machine by making use of hashing for speed:

LEMMA 1. The time complexities of functions "tcons", "car" and "cdr" on <TUP> are all O(1).

(2.2) A function "settup" transforms a <TUP> into a <SET> with the corresponding elements; "tupset" does the converse, e.g.,
settup[(A,B)]={A,B} or {B,A};
settup[(A,B,B)]={A,B} or {B,A};
tupset[{A,B}]=(A,B) or (B,A).
Specially for t ∈ TUP, tupset[t]=t (a coercion rule). Although the ordering of elements of a <SET> is irrelevant to its identity, the ordering of the elements of the <TUP> used first to define a <SET> establishes a "canonical order" among the elements of the <SET>. Whenever the canonical order is needed, it can be retrieved by performing tupset [<SET>]. <SET>'s are represented uniquely in the machine by making use of hashing for speed:

LEMMA 2. For t ∈ TUP, s ∈ (SET ∪ TUP), the time complexities of settup[t] and tupset[s] are O(|t|) and O(1), respectively.

(2.3) For x ∈ ID the function "ass" yields an <ASS> : ass[x]=(.x*). (* means actual datum represented by the variable). Conversely, for a=ass[x] ∈ ASS the function "key" gives the <ID>, x: key[a]= x and the pseudo-function assign[a;v] assigns a value v, of any type, to <ASS>, a. The value is assign[a;v]= v and the assigned value can be retrieved as the value of the function value[a]=v. The initial value of an <ASS> is (). Similarly to Lisp, property functions are defined as put[x;y;v]= assign[ass[tup [x;y]];v], get[x;y]=value[ass[tup[x;y]]] and remprop[x;y]=put[x;y;()], where x, y ∈ ID and v is a datum of any type. These functions are implemented

by making use of hashing for speed:

LEMMA 3. The time complexities of "ass", "key", "assign", "value", "put", "get" and "remprop" are all O(1).

Note in ordinary Lisps that properties are more restrictive: x ∈ SYM and y ∈ (INT ∪ SYM), and that in case *m* properties are used on a SYM the time complexity may increase as *O(m)* due to list implementation of properties.

(2.4) For x, y ∈ ID, the predicate function eq[x;y] checks the equality of x, y in accordance with the mathematical common sense. Namely, in case x and y are of different types, eq[x;y]=(); for x, y ∈ INT, eq[x;y]=T iff x and y are numerically equal; for x, y ∈ SYM, eq[x;y]=T iff x and y have the same spelling; for x, y ∈ ASS, eq[x;y]=T iff key[x]=key[y]; for x, y ∈ TUP ∪ SET, eq[x;y]=T iff x and y represent the same n-<TUPle> or <SET> mathematically. E.g.,

eq[(A,B);(B,A)]=(), eq[{A,B};{B,A}]=T,
eq[{A,B};{B,B,A}]=T, eq[(.(A));(.{A})]=().

LEMMA 4. The time complexity of "eq" is O(1).

Note that for the equality checking of Lisp data <CONS>, the time consuming function "equal" has to be used[3]. <TUPle>'s essentially differ from <LIST>'s in this regard.

(2.5) Outline of an Implementation called HLISP (Hashed LISP).

Each <HLISP CELL> in the FSA (Free Storage Area) consists of three fields: <CELL> ::= [<TAG>,<CAR field>,<CDR field>]$_o$. Besides for GBC (GarBage Collection) marking, the <TAG> is used to specify the data type of the cell. Similarly to Lisp 1.5, a <CONS CELL> ::= [CONS,x*,y*]$_o$ is created in the FSA as the result of cons[x;y]. The FSA itself is used as the (only one) hash table with the size being a prime p. For tup[x;y], a hash search (insert iff absent) is made for a <TUP CELL> ::= [TUP,x*,y*]$_o$, using Knuth's algorithm D[4, p521], thereby ensuring uniqueness of the resultant <TUP>. For ass[x], a hash search is made for an <ASS CELL> ::= [ASS,"don't care",x*]$_o$, using Knuth's algorithm U2[4, p539]. The value of the <ASS> is placed in the <CAR field>, which is not used as the key of the hash search. A <Short INTeger> is represented as a pointer (placed in <CAR> or <CDR> field) to a non existing memory address. An n-precision <INT> is uniquely represented like a <TUP> of <Short INT>'s (i_1, i_2, ..., i_n) with the head cell being changed to an <INT CELL> ::= [INT,i_1,t]$_o$, where t is a <TUPle>, (i_2, ..., i_n). A <SYM CELL>, corresponding to an atom header cell of Lisp 1.5, is the same as an <INT CELL>, except the head cell <SYM CELL> ::= [SYM,i_1,t] with <Short INT>'s i_1, ..., i_n being an unique encoding of the character string which identifies the <SYM>. For settup[t], t=(e_1*, ..., e_m*), a <SYS1 CELL> ::= [SYS1, "don't care", "don't care"]$_o$ is made first, where SYS1 is a system data tag. Secondly, a <TUP> t'=(e_1'*, ..., e_n'*), free of duplicating elements is made from t by using hash searches for <SYS2 CELLs> ::= [SYS2, "pointer to the SYS1 cell", e_i*]$_o$ for removing duplications with time complexity *O(1)* per element of t. Thirdly, using a symmetric (in respect to permutation of arguments) hash sequence h_i(e_1'*, ..., e_n'*) i=1, 2, 3, ... (e.g., h_1=mod(e_1'*+ ... + e_n'*,p-1)+1, h_i=mod(i*h_1,p) with time complexity *O(n+i)*; Algorithm U2[*ibid*], [5]), hash search is made for a cell s=[SET,h_1, "don't care"]. If unsuccessful, a new <SET CELL>, <SET CELL> ::= [SET,h_1,[SYS1,|s|,t']]$_o$, is created. If successful, s = settup[t] (redefined <SET>) or ≠ (hash conflicting <SET>'s) is checked by utilizing the <SYS2 CELL>'s of t. (Time complexity *O(|t'|)* at the most.) The hash search is resumed in the latter case. The load factor α of the FSA is limit to α≤α*M*<1 (e.g., α*M*=80%). When α≥α*M* the GBC is called. A trioccupancy ("occupied" (i.e., a cell in use), "deleted" (not in use but in hash conflict) and "empty" (neither in use nor in conflict)) scheme is used to reclaim the garbage <CELL>'s without cell relocations and without using secondary storage. (A detailed analysis is given in [6]; McCarthy [7], proposed a scheme essentially the same as the present uniquely represented n-<TUPles>. However, he stated a difficulty in GBC: the neccesity of the use of secondary storage.) If the result of GBC does not satisfy α<α*m* (e.g., α*m*=60%), GGBC (Grand GBC; more details are given in IV) is called. If α<α*m* is still not satisfied the job is terminated because of insufficient storage. Note that the condition α*m*<α*M*<1 ensures the time complexities as claimed in LEMMAs 1-4. If α*m*=α*M*=1 were used, the FSA would be usable up to the very last one cell, but the LEMMAs would not be valid.

III. Application to Formula Manipulation.

Let IP be the set of polynomials with integer coefficients and positive integer exponents.

(3.1) The <Sum of Product> Normal Form.

Polynomials of IP can be expressed as sum of products (terms), e.g.,

p1 = $2UV^2 + 3X^3Y^4$, p2 = $3Y^4X^3 + VUV + UV^2$.

These expressions represent the same polynomial, and they can be faithfully represented in terms of <TUP>'s as follows:

<SP*form>::=((<TERM ID*>,<COEFficient>),...)$_o$ and
<TERM ID*>::=((<VARiable ID>,<EXPonent>),...)$_o$,

where <COEF> ∈ INT0, <VAR ID> ∈ SYM and <EXP> ∈ INT+. E.g.,

sp*(p1)=((((V,2),(U,1)),2),(((X,3),(Y,4)),3))
sp*(p2)=((((Y,4),(X,3)),3),(((V,1),(U,1),(V,1)),1),
(((U,1),(V,2)),1)).

These SP* forms can be transformed into a unique SP normal form in the following way (a program is given later): (1) Combine duplicating <VAR ID>'s in a <TERM ID*> as in VUV=V^2U. (2) Absorb the commutative nature of multiplications into a SET: <TERM ID> ::= {(<VAR ID>,<EXP>),...}$_o$. E.g., $V^2U=UV^2$ is absorbed as {(V,2),(U,1)}={(U,1),(V,2)}. (3) Combine duplicating <TERM ID>'s as in $V^2U+V^2U=2V^2U$. (4) Absorb the commutative nature of additions into a SET: <SP> ::= {(<TERM ID>,<COEF>),...}$_o$. E.g.,

sp[p1]=sp[p2]={({(V,2),(U,1)},2),({(X,3),(Y,4)},3)}.

We now define two data structures, in order to formalize the definition of the <SP> form:
A <CLUB> is a <SET> of 2-<TUPle>'s of <ID>'s (informally, <CLUB> ::= {..., (m*i*, g*i*), ...}$_o$) such that all of the first element, to be called the (club-) "member", of the 2-<TUPle>'s are distinct (m*i*≠m*j* for *i*≠*j*). The second elements (g*i*'s) of the 2-<TUPle>s are called "grade"s. A <MULTISET> is a special <CLUB> of which the grades are restricted to positive integers. (This agrees with the "multiset" of Knuth[4] by regarding the "multiplicity" as the grade.) Thus, we can now state: "An <SP> is a <CLUB> of <TERM ID>'s with non-zero integer grades, called <COEF>; a <TERM ID> is a <MULTISET> of <SYM>'s, called <VAR ID>'s; specially, for the null and constant polynomials,

sp(0)={}, sp(n)={({},n)}, where n ∈ INT0."

Since the SP form obviously represents IP polynomials uniquely, i.e., for p, q ∈ IP,
sp(p) = sp(q) (set equality) iff p-q ≡ 0,
by LEMMA 4 we obtain:

PROPOSITION 1. Given two IP polynomials in the SP form, the time complexity for identity checking of the two is O(1).

(3.2) Polynomial Manipulation in The SP Form:
A Property Adding Auxiliary Function:

```
addprop[g;x;v;r] = prog[[y];y:=get[g;x];
   [null[y] → prog2[put[g;x;v];r:=tcons[x;r]];
    T → put[g;x;v+y]];return[r] ].
```

Given g, x ∈ ID, v ∈ INT and r ∈ TUP, if the G-property (i.e., the value of get[g;x]) is (), "addprop" puts v on the property and appends x to r in the result, otherwise, v is added into the property. By LEMMAs 1 and 3, the time complexity is *O(1)*. Similarly, we define:

```
subprop[g;x;v;r]=addprop[g;x;-v;r].
```

A Property into Club-Grade Function:

```
clubprop0[g;r]=prog[[c;y;w];w:=r;
 A [null[w] → return[settup[c]];y:=get[g;car[w]];
    [y≠0 → c:=tcons[tcons[car[w];tcons[y;()]];c]] ;
    remprop[g;car[w]];w:=cdr[w];go[A] ].
```

Given g ∈ ID and r, a <TUPle> of distinct <IDs>, "clubprop0" yields a club of the <IDs> with making the respective G-properties into grades and excluding 0-grade members. By LEMMAs 1, 2 and 3 and since loop A is executed |r| times, the time complexity is *O(|r| + 1)*. *1* is added to account the time *O(1)* needed in case |r| = 0, i.e., r = ().

A Club Union and Grade-Adding Function:

```
addclub[p;q]=prog[[g;r;w];g:=gensym[];w:=tupset[p];
 A [null[w] → prog2[w:=tupset[q];go[B]]];
  r:=addprop[g;caar[w];cadar[w];r];w:=cdr[w];go[A];
 B [null[w] → return[clubprop0[g;r]]];
  r:=addprop[g;caar[w];cadar[w];r];w:=cdr[w];go[B]].
```

Given clubs p, q with numerical grades, "addclub" yields a club of the union of members of p and q with the grades of common members being added in and 0-grade members being excluded from the result. A "gensym" (i.e., a unique <SYM> generated by the system) is used to avoid possible confusions of properties in the auxiliary functions. Similarly, subclub[p;q] is defined by replacing the "addprop" in the last line only by "subprop". Since loop A is repeated |p| times and loop B, |q| times and by LEMMAs 1, 2 and 3, the time complexity is *O(|p|+|q|+1)*. In case p, q ∈ SP "addclub" adds the two and gives the result in the SP normal form. Hence,

PROPOSITION 2. The time complexity of adding two polynomials p and q in the SP form is O(|p|+|q|+1). (Multivariateness has no effect.)

A Polynomial Multiplier Function:

```
mulsp[p;q]=prog[[g;r;u;v];g:=gensym[];u:=tupset[p];
 A [null[u] → return[clubprop0[r]]];v:=tupset[q];
 B [null[v] → prog2[u:=cdr[u];go[A]]];
   r:=addprop[g;addclub[caar[u];caar[v]];
              cadar[u]*cadar[v];r];v:=cdr[v];go[B]].
```

Given p, q ∈ SP, "mulsp" yields the product in the SP form. Note that "addclub" is used to multiply two <TERM ID>'s as in addclub[{(A,1),(B,2)};{(B,3),(C,4)}]={(A,1),(B,5),(C,4)}. For s ∈ SP, let T(s) = |s|+ (total number of elements in <TERM ID>'s of s). The dominating term (clubprop0[r] is *O(|p|·|q|)* at the most) in the time complexity of "mulsp" is easily seen to be *O(|q|T(p)+|p|T(q))*, which arises from repeating the "addclub" on <TERM ID>s for |p|·|q| times in the nested loops A and B. Hence, we obtain:

PROPOSITION 3. The time complexity of multiplying p, q ∈ SP is O(|q|T(p)+|p|T(q)); specially in case each term is K-variate at the most, it is O(|p|·|q|(K+1)) and in the univariate case it is O(|p|·|q|). (Factors such as $\log_2|p|$ or $\log_2|q|$ are absent. Sparseness of the result has no effect.)

An SP* into SP Transformation Function:

intosp[p]=mulsp[p;{({},1)}]$_\circ$, where {({},1)}=sp(1).

This works correctly because of the "coercion rule" in (2.2). Let T*(p)=|p|+(total number of elements in <TERM ID*>s of p ∈ SP*). We obtain:

PROPOSITION 4. The time complexity of transforming an IP polynomial p in an SP form into the SP normal form is O(T*(p)); specially in case the length of each term of p is K at the most, it is O(|p|·(K+1)). (If <TERM ID*> and SP* were sorted into a sorted normal form, the time complexity would be O(|p|·($\log_2|p|$)·(K+1)$\log_2$(K+1)).)*

(3.3) The <Signed Absolute SP> form:
Let s=sp(p) be the SP form of a polynomial p ∈ IP. As a <SET>, s can be partitioned uniquely as s = s+ ∪ s-, wherein all grades of s+ are positive and those of s-, negative. Let -s- be the <SP> obtained by reversing all signs of grades of s-.

Definition. The <Absolute SP> form asp(p) of p is a <SET>: asp(p)={s+,-s-}; specially asp(0)={}.

PROPOSITION 5. For p, q ∈ IP,
asp(p) = asp(q) iff (p ≡ q ∨ p ≡ -q).

Definition. The <SASP> normal form sasp(p) of p is a 2-<TUPle>: sasp(p)=(asp(p), sign(p)), where sign(p)=+1 in case the canonical order of the SET , asp(p) is tupset[asp(p)]=(s+,-s-), otherwise sign(p)=-1 (c.f., (2.2)); specially, sasp(0)=().

PROPOSITION 6. For p, q ∈ IP,
sasp(p) = sasp(q) iff p ≡ q.

(3.4) Unique Normal Forms for Rationals:
Let Q be the set of (all) rational numbers. Hereinafter, for q ∈ Q, we use the following obviously unique representation; if q ∈ INT ⊂ Q use the integer q itself; otherwise use the 2-<TUPle>, (a*,b*) such that a, b ∈ INT, b≥2, q=a/b and a, b are relative primes.

SP, ASP and SASP forms can be easily generalized to <QP, polynomials with rational coefficients and positive integer exponents> by changing the condition <COEF> ∈ INT0 for <IP>'s into <COEF> ∈ (Q - {0}).

Let QF be the set of rational functions with rational coefficients and integer exponents, i.e., QF={x/y| x ∈ QP, y ∈ (QP - {0})}. Any function r ∈ (QF - {0}) is known to be uniquely factorizable, except the arbitrariness of signs on the factors, as follows:

$$r = q\, p_1^{e_1} \cdots p_i^{e_i} \cdots p_k^{e_k},$$

wherein q ∈ (Q - {0}), e_i ∈ INT0 and p_i ∈ (IP - INT) such that p_i is not factorizable into elements of (IP - {-1,1}).

Definition. The <Factorized SASP> form fsasp(r) of r ∈ (QF - {0}) is a 2-<TUPle>:

fsasp(r) = ({..., (asp(p_i),e_i), ...}, $\underline{+q}$), where

$$\underline{+q} = (\text{sign}(p_1))^{e_1} \cdot\cdot (\text{sign}(p_i))^{e_i} \cdot\cdot (\text{sign}(p_k))^{e_k} \cdot q;$$

specially, fsasp(0)=().

PROPOSITION 7. For x, y ∈ QF,
fsasp(x) = fsasp(y) iff x ≡ y.
PROPOSITION 8. For x, y ∈ (QF - {0}),
car[fsasp(x)] = car[fsasp(y)] iff x/y ∈ Q.

Proofs of PROPOSITIONs 5 to 8 have been omitted but they would be easy.

A Multiplier for x, y ∈ (FSASP - {()}):

```
mulfsasp[x;y] = tcons[addclub[car[x];car[y]];
                      tcons[mulq[cadr[x];cadr[y]];()]],
```

where "mulq" is a multiplication function of rational numbers. For a divider "divfsasp", replace "addclub" by "subclub" and "mulq" by a rational number divider "divq".

(3.5) Poisson series is a function as:

$$p = \sum_i a_i \cos(u_i) + \sum_j b_j \sin(v_j),$$

where $a_i, u_i, b_j, v_j \in \underline{QF}$.

A unique normal form POIS for this series can be obtained by absorbing the arbitrariness caused by cos(u) = cos(-u) and sin(v) = -sin(-v) into ASP forms: <POIS> ::= (<POIS COS>,<POIS SIN>)$_\circ$, wherein <POIS COS> and <POIS SIN> are clubs:

<POIS COS> ::= {(asp(u),sp(a)),...}$_\circ$ and
<POIS SIN> ::= {(asp(v),sp(sign(v)b)),...}$_\circ$

with u ∈ QF and a, b, v ∈ (QF - {0}). It would be a matter of exercise to define Lisp functions to perform addition, subtraction and multiplication on POIS normal forms.

(3.6) The <Associator List SP> Form:

So far stress has been laid on unique normal forms and on time complexities. However, for improvements in actual speed of computation, constant factors neglected in time complexities must be taken into account. Although time complexities of cons[x;y] and tcons[x;y] are both *O(1)*, "cons" would actually work faster than "tcons" because of extra hashing overhead time needed in "tcons" to ensure uniqueness. Similarly, "value", "key" and "assign" would be faster than "ass" (c.f., (2.3)). The same would hold for the *O(n)* complexity for list[x_1; ...;x_n] and settup[t] with |t|=*n*. It would be a reasonable strategy to use unique normal forms only where they are essentially needed. For example, in the manipulation (add, sub and multiply) of <IP>'s in the SP form, use of the unique normal forms for <TERM ID>'s is essential but use of a <SET> for sum of terms is not. Use of the following ALSP form would be better for the sake of speed: <ALSP> ::= (Ø(.(g*,<TERM ID>)),... Ø)$_\circ$. For p ∈ IP, alsp(p) is a <LIST> of <ASSociator>'s of 2-<TUPle>'s of a "gensym", g* and a <TERM ID>. <COEF>'s of the sp(p) are given as G-properties (i.e., get[g*;<i-th TERM ID>] = <i-th COEF>). Rewriting functions for SP forms in (3.2) into those for ALSP forms would be a matter of exercise. The similar applies to Poisson series: Use ASP forms for u's and v's and ALSP forms for a's and b's.

IV. Computing Schemes with Reclaimable Hash Tables

The choice between tabulation and recomputation is a basic problem in programming. While (hashed) tabulation provides the best time complexity of *O(1)* in many cases, extra storage space is needed to keep the tables.

In HLISP two features called tabulative and associative computing are provided, which enable users to utilize the full advantages of computing with hash tables. Moreover, in order to make a compromise between the space and time requirements automatically, a two staged garbage collection scheme, GBC and GGBC of (2.5), is employed. The <CELL>'s used for hash table entries in "tab-" and "assoc-comp" schemes are reclaimed by GGBC but not by GBC. Hence, these entries are termed "reclaimable". After having been reclaimed, the table entries are reconstructed on demand.

(4.1) "Tabcomp" is applied to member[x;s]=(x ∈ s) for x ∈ ID, s ∈ SET and to n-way switching and selecting functions: tabαβ[x;a;e*] with α ∈ {a,d,q,g} and β ∈ {q,g} . The value of a must be an n-<TUPle> of the form a=(..., (*mi**, *gi**), ...) and e* must be a constant <ID> datum. If x matches with *mi* (∈ ID), the resultant value is respectively cadr[(*mi**, *gi**)]=*gi**, cdr[(*mi**, *gi**)]=(*gi**) or (*mi**, *gi**) for α=a, d or q; for α=g the result is "GO TO *gi**". If no match, for β=q the resultant value is e* and for β=g the result is "GO TO e*".

(4.2) "Assoccomp" effectively avoids the recomputation of the same function for the same argument(s) by inserting the results of the previous computation in the reclaimable hash table entries. Evaluation of a function is made in the "assoccomp" mode by so specifying to the compiler or interpreter. By "assoccomp", the time complexity of recursive algorithms such as follows can be improved automatically without rewriting.

```
factorial[n]=fc[n]=[n=0 → 1;T → n*fc[n-1]],
fibonacci[n]=fb[n]=[n≤1 → n;T → fb[n-1]+fb[n-2]],
```

$_nC_m$=c[n;m]=[m=0 ∨ m=n → 1;T → c[n-1;m]+c[n-1;m-1]].

(4.3) *LEMMA 5. Time Complexities of Tab- and Assoc-comp features are as in the following table:*

	WITHOUT Tab- and Assoc-comp features.	WITH Tab- or Assoc-comp		
Function	TIME	INITIAL TIME	REPEATED TIME	EXTRA CELLS
member[x,s]	$O(\|s\|)$	$O(\|s\|)$	$O(1)$	$\|s\|$
tabαβ[x,a,e*]	$O(\|a\|)$	$O(\|a\|)$	$O(1)$	$\|a\|+1$
factorial[n]	$O(n)$	$O(n)$	$O(1)$	$2n+3$
fibonacci[n]	$O(1.618^n)$	$O(n)$	$O(1)$	$2n+3$
$_nC_m$=c[n,m]	$O(_nC_m)$	$O(n^2)$	$O(1)$	$3n^2/2$

The initial time means the time complexity immediately after a GGBC call. Extra cells are the number of <CELL>'s needed for reclaimable hash entries. E.g., repeated evaluation of fb[21]=10946 runs 30,000 times faster in HLISP by merely feeding a card "ASSOCCOMP ((FB))". clubmember[x;c]= tabqq[x;tupset[c];()] checks whether x is a member of the <CLUB>, c. The time complexity of $O(|s|\cdot|t|)$ in the pure Lisp algorithms[3] for s ∪ t and s ∩ t of sets s, t is greatly improved by applying "tabcomp" to "member" (even immediately after a GGBC call):

LEMMA 6. Time complexity of s ∪ t and s ∩ t for s, t ∈ SET is $O(|s|+|t|)$.

(4.4) Outline of an HLISP Implementation:

For "member" <SYS2 CELL>'s of (2.5) are utilized. When <SYS2 CELL>'s are reclaimed by GGBC, the <SYS1 CELL> is switched to a <SYS1* CELL> to indicate the necessity of reconstruction of the <SYS2 CELL>'s. For "tabαβ", initially (i.e., after GGBC) a <SYS3 CELL> ::= [SYS3,a*,e*]$_\circ$ is hash inserted (as a result of an unsuccessful search) and then

<SYS4 CELLs> ::= [SYS4,(mi*,gi*),[SYS3,a*,e*]]$_0$ are hash inserted by using a hash sequences determined by mi's (not the <TUP> (mi, gi)) and the pointer to the <SYS3 CELL>. Hash retrieval is made by utilizing these <SYS3 CELL> and <SYS4 CELL>'s, which are all reclaimed by GGBC. In the assoccomp mode, a function fb[n], say, is evaluated as: First, make a hash search for <SYS5 CELL> ::= [SYS5, "don't care", t]$_0$, with t=tcons[n;FB], and if unsuccessful insert a <CELL>, [SYS5,1*,t], where 1* is a <SYStem SYMbol>, then compute fb[n] and replace 1* by fb[n] for future retrieval of fb[n]. Else if successful retrieve the value from the <CAR field>. Specially, in case the <CAR field> contains 1*, there must have been a vicious circle in the algorithm such as fb[n]=[n≤1 → n; T → fb[n]+ fb[n-1]]. Thus a message "CIRCULAR DEFINITION ERROR IN FB ..." is printed. GGBC reclaims <SYS5 CELL>'s except those containing 1*. Hence,

LEMMA 7. "Assoccomp" effectively checks circular definitions at runtime.

(4.5) For fc[n], fb[n], c[n,m] etc., "assoccomp" is more convenient than "tabcomp" since the range of argument(s) is generally not known in advance. Conversely, if "assoccomp" were used for member[x;s], say, a great number of wasteful hash entries for x ∉ s would be created. Thus, "tab- and assoc-comp" are complementary and each has its own raison d'être.

V. Concluding Remarks

The first version of HLISP without the SET feature has been in operation for two years[8], but with the TUP feature alone little advantage in formula manipulation could be found. The combination of SETs and TUPs is believed to have provided a really powerful tool for formula manipulation as indicated in III. Tab- and assoc-comp features would also be useful. Since the implementation of efficient hashing and garbage collection algorithms is a very specialized art, it would be better to separate them from the general users. Therefore, external specifications of such algorithms have been given as LEMMAs in this paper.

The following improvements are now in progress to make the schemes presented in this paper into truly useful tools for symbolic and algebraic computations:

(1) Writing of an efficient HLISP compiler[9].
(2) Implementation of a language system called "FLATS" which would enable us to absorb any existing algorithm written in Fortran, Lisp or Algol 60; and to write new algorithms with Tuples and Sets added to any of the three languages F, L or A, whichever the user may prefer (HLISP = FLATS).
(3) Design of hashing, GBC and runtime type check hardware to improve the ultimate speed of "FLATS".

The authors acknowledge Messrs. M. Terashima[10] and F. Motoyoshi[9] for their valuable contributions in implementing HLISP.

VI. References

[1] S.C. Johnson, *SIGSAM Bulletin*, 8, 3, p.63, '73.
[2] E. Horowitz, *J. ACM*, 22, 4, p.450, 1975.
[3] J. McCarthy, et al., *LISP 1.5 Programmer's Manual*, MIT press.
[4] D.E. Knuth, *The Art of Computer Programming, vol. 3*, Addison-Wesley, Reading, Mass., '73.
[5] M. Sassa and E. Goto, *A Hashing Method for Fast Set Operations*, submitted for publication.
[6] T. Gunji, *Tech. Rep. 76-03, ISD* (Information Science Department, the University of Tokyo), 1976.
[7] J. McCarthy, Page 151 of *Symbol Manipulation Languages and Technique*, D. Bobrow, ed., North-Holland, 1971.
[8] Y. Kanada, *Tech. Rep. 75-01, ISD*, 1975.
[9] F. Motoyoshi, *Tech. Rep. 76-05, ISD*, 1976.
[10] M. Terashima, *Tech. Rep. 75-03, ISD*, 1975.
[11] A.C. Hearn, *REDUCE2 User's Manual, 2nd. ed.*, Salt Lake City, Utah., 1973.

APPENDIX. Actual Timing Data for Polynomial and Poisson Series Manipulations.

REMARKS: (1) The machine used is HITAC 8800/8700 at the Computer Centre of the University of Tokyo.
(2) The same HLISP interpreter system was used as the host system for REDUCE 2[11]. The free storage area was 75K cells in which 25K cells were reserved for <ID> objects.
(3) The data for polynomial multiplication were obtained to observe the dependence of time on n (number of terms in polynomials) and multiplicity, K. Observed times were normalized by $n^2(K+1)$ as PROPOSITION 3 predicate. Unit of time is in msec. '*' means 'not measured'.
(4) The FORTRAN data of univariate case were taken by a program with explicit code for hashing. The program is similar to the algorithm by Gustavson and Yun to be given at this SYMSAC '76. The hash area was selected to 5011 (a prime) and the hash probe sequence was given by Algorithm U2 of Knuth[4, p539].
(5) The programs in HLISP were written for the ALSP and ASP forms of (3.6).

Formulas \ n t=resultant # of terms	4	8	16	32	4	8	16	32	4	8	16	32	
$(\sum_{i=1}^{n} A^i) * (\sum_{j=1}^{n} A^j)$ $t=2n-1$	1.71	1.69	1.60	1.60	1.85	1.73	1.71	1.67	1.82	1.74	1.74	1.77	← HLISP
	4.42	2.95	3.97	5.45	3.67	3.50	4.43	7.20	4.65	4.04	5.54	9.10	← REDUCE
	.025	.024	.020	.016									← FORTRAN
$(\sum_{i=1}^{n} A^i) * (\sum_{j=1}^{n} A^{jn+1})$ $t=n*n$	1.76	1.74	1.72	1.73	1.98	1.78	1.76	1.80	1.81	1.80	1.79	1.84	← HLISP
	5.50	6.08	15.4	51.3	4.33	7.37	21.6	*	4.40	8.48	*	*	← REDUCE
	.025	.028	.020	.018									← FORTRAN
$(\sum_{i=1}^{n} A^{-2+3i}) * (\sum_{j=1}^{n} A^{-3+4j})$ $t=7n-12$	1.96	1.71	1.68	1.63	1.88	1.82	1.73	1.74	1.84	1.83	1.79	1.77	← HLISP
	5.35	5.85	8.20	14.3	5.42	6.53	10.6	*	5.16	7.64	12.2	*	← REDUCE
	.028	.025	.020	.016									← FORTRAN
K-variate	1-variate (A=X)				2-variate (A=XY)				4-variate (A=XYZU)				

Timing Data for Poisson Series Manipulation:	HLISP	REDUCE
(A1*COS(WT)+A3*COS(3*WT)+B1*SIN(WT)+B3*SIN(3*WT))**3	1587 msec	8077 msec

A Machine Independent Implementation of a Collection Algorithm for the Multiplication of Group Elements

Volkmar Felsch
Lehrstuhl D für Mathematik
Rheinisch-Westfälische Technische Hochschule
Aachen, Germany

1. Introduction

For any finite soluble group G there exists a system of generators $g_1,\ldots,g_n$ such that each element g of G can be expressed uniquely as a "normed word" $g_n^{\varepsilon_n}\ldots g_1^{\varepsilon_1}$ with $0 \le \varepsilon_i < p_i$ for $1 \le i \le n$, where $p_1,\ldots,p_n$ are appropriate non-negative integers such that $\prod_{i=1}^{n} p_i = |G|$. There are well known "collection algorithms" which compute the corresponding normed word for any given word $g_{i_1}^{\lambda_1}\ldots g_{i_m}^{\lambda_m}$ in the generators $g_1,\ldots,g_n$, provided that certain rules for this process are given by a set of defining relations for G.

The collection algorithms have found several important applications. Probably the best known of these is the "nilpotent quotient algorithm" which computes the maximal p-factor group of a given (not necessarily finite or soluble) group for some prescribed prime p. Originally suggested by I. D. Macdonald [11] and J. W. Wamsley [15, 1], it has been further investigated during the last few years and found some remarkable applications (see e. g. [3,6,14]).

On the other hand, such a collection algorithm can be used to compute the product gh or the inverse g^{-1} for any arbitrary elements $g = g_n^{\varepsilon_n}\ldots g_1^{\varepsilon_1}$ and $h = g_n^{\delta_n}\ldots g_1^{\delta_1}$ of G by collecting the word $g_n^{\varepsilon_n}\ldots g_1^{\varepsilon_1} g_n^{\delta_n}\ldots g_1^{\delta_1}$ or the word $g_1^{|g_1|-\varepsilon_1}\ldots g_n^{|g_n|-\varepsilon_n}$, respectively, hence the knowledge of a suitable presentation of G suffices to compute the local structure of G even in cases where no concrete representations of the group elements (e. g. as permutations or as matrices over some field) are known. Moreover, even if a concrete representation is known, it may well be that the use of a collection algorithm saves a lot of space and computation time.

Programs for such collecting multiplication algorithms (cm-algorithms) have been written by J. Neubüser [12], W. Lindenberg [9,10], H. Jürgensen [7] and E. Geller [5]. These implementations (except the last one which constructs partial multiplication tables) use sophisticated preprocessors in order to generate an individual machine code collection routine for any given presentation. Hence they are extremely machine dependent, and at the end of the sixties each of them had gone with the computer for which it had been designed, leaving behind a badly felt gap and the need for a machine independent successor.

It is the purpose of this paper to fill this gap by presenting a machine independent, portable implementation of such a preprocessor. Moreover, I shall discuss the impact of variations on the efficiency of the cm-algorithm and its behaviour for unfavourable presentations. For the reader's convenience these two sections will be preceded by a short summary of the essential features of the previous algorithms. Finally, the last section will give those technical details which are of interest only for the expert.

The program to be described has recently been implemented in the Aachen-Sydney Group System.

2. The previous implementations

In the following we shall assume that the group G is given by a power-commutator presentation (pc-presentation) of the form

$$G = \langle g_1,\ldots,g_n \mid g_i^{p_i}=g_{ii},\ [g_i,g_j]=g_{ij} \text{ for } 1\le i\le n \text{ and } i<j\le n\rangle$$

which satisfies the following conditions. Let $G_i = \langle g_1,\ldots,g_i\rangle$ for $1 \le i \le n$, and let G_0 be the identity subgroup of G, then for $1 \le i \le j \le n$

(i) $1 < p_i \in \mathbb{Z}$ and $g_i^r \notin G_{i-1}$ for $1 \le r < p_i$,

(ii) g_{ij} is a word in the generators $g_1,\dots,g_{j-1}$.

Such a pc-presentation exists if and only if G is a finite soluble group.

The cm-algorithms used by the previous implementations are all based on the same simple collection process (known as "collection from the right") which computes the normed word $\overline{w}$ for any given word $w = g_{i_1}^{r_1}\cdots g_{i_m}^{r_m}$ with non-negative exponents by applying the following rule:

If w is not yet a normed word, then there is a maximal k such that $r_{i_k} \ge p_{i_k}$ or $i_k \le i_{k+1}$. For simplicity assume $g_{i_k}^{r_k} = g_i^r$ and $g_{i_{k+1}}^{r_{k+1}} = g_j^s$ (if $k < m$). If $r \ge p_i$, then replace the factor g_i^r by $g_i^{r-p_i} g_{ii}$, else replace the product $g_i^r g_j^s$, and that by g_i^{r+s}, if $i = j$, or by

$$(1) \qquad g_i^{r-1} g_j g_i g_{ij} g_j^{s-1},$$

if $i < j$. Then drop all factors with exponent zero and repeat the process for the newly constructed word.

It can easily be proved that this algorithm yields the normed word $\overline{w}$ after a finite number of steps and that the intermediate words will not exceed a certain length. But unfortunately, this method is rather inefficient if it is implemented as a general procedure which is applicable to any pc-presentation. This is mainly due to the fact that the words g_{ij} are applied and the occurring exponents are checked repeatedly even if they are trivial.

As mentioned above, it is a characteristic feature of the three previous programs that they use a preprocessor to overcome this difficulty. For any given presentation such a preprocessor constructs in a kind of initializing phase an individual collection routine which in its structure involves all information from the given presentation and hence is able to avoid making further reference to the defining relations.

Besides this most essential feature the three programs speed up the algorithm by improving the algorithm itself. Instead of replacing a product $g_i^r g_j^s$ by (1) they use more favourable words which, however, require the knowledge of further relations.

Jürgensen, for instance, computes a collection routine as described above, but he only uses it to calculate the normed words $\overline{g_{ijs}}$ for all commutators $g_{ijs} = [g_i, g_j^s]$ with $1 \le i < j \le n$ and $1 \le s < p_j$. Then he further improves his program by constructing a final version of the collection routine which replaces any product $g_i^r g_j^s$ by

$$(2) \qquad g_i^{r-1} g_j^s g_i \overline{g_{ijs}}.$$

Lindenberg's program assumes all relations of the form $g_i g_j^s = \overline{g_i g_j^s}$ to be given as input and uses the word

$$(3) \qquad g_i^{r-1} \overline{g_i g_j^s}$$

instead of (1).

3. A machine independent implementation

The cm-algorithms described in section 2 are part of program systems which were entirely written in the internal code of the respective computers. So the implementation of a preprocessor generating machine code raised no technical difficulties.

The situation was different, however, when we decided to build a cm-algorithm into the Aachen-Sydney Group System. This is a large FORTRAN-written, portable system for the investigation of groups [2,4]. Therefore it was our main problem to find a suitable machine independent way to realize the algorithm.

It is an obvious possibility (and certainly a practical one, because it yields a fast, internally coded collection routine) to formulate the preprocessor and the collection routine as machine code functions in the FORTRAN program. But this would be incompatible with the portability of the system, and so we rejected it.

Another idea which reduces the machine dependent part and yet provides a machine code collection routine is the following. In a first step a FORTRAN-written preprocessor is used to formulate a cm-algorithm in some properly chosen, standardized code, and then in a second step the routine constructed is translated into internal code by the aid of an appropriate translator. There are portable versions of such translators available which have been designed for much more general purposes but in particular are able to do this job provided that the underlying tables which contain the information about both codes (and may be very large) are set up accordingly.

The disadvantages of this second idea are the possibly disproportionate size of

instruction	meaning
1,ℓ	call ℓ
2,ℓ	goto ℓ
3,j,ℓ(0),...,ℓ(p_j-1)	goto ℓ(δ_j)
4,ℓ(1),...,ℓ(p-1)	goto ℓ(ε) [ε is a global variable]
5,i,ℓ	if $\varepsilon_i > 0$, set $\varepsilon = \varepsilon_i$ and call ℓ
6,i,ℓ	if $\varepsilon_i > 0$, set $\varepsilon = p_i - \varepsilon_i$ and call ℓ
7,i,ℓ	if $\varepsilon_i = 0$, goto ℓ; else set $\varepsilon = p_i - \varepsilon_i$
8	set $\delta_j = 0$ for $j = 1,\ldots,n$
9	set $\delta_1 = \mathrm{mod}(\delta_1, p_1)$; end
10,m	set $\delta_1 = \delta_1 + m$ and return
11,j,m	set $\delta_j = \mathrm{mod}(\delta_j + m, p_j)$ and return
12,j,m	set $\delta_j = \mathrm{mod}(\delta_j + m, p_j)$; if $\delta_j \geq m$, return

Table 1. Special code for the collection routine

the translator and again the machine dependence. Nevertheless it leads us to a successful solution: If one drops the second step and adds instead a FORTRAN-written interpreter to the program which is able to work off the special code routine constructed in the first step, then one has a completely machine independent collection routine. On the other hand this method still leaves the freedom to apply the second step in order to get a faster routine whenever the required data for a suitable translator are available.

The implementation to be described realizes this idea.

Our first task was to fix the special code. Here we assumed that, whenever the collection routine is called via the interpreter, two elements $g = g_n^{\varepsilon_n} \cdots g_1^{\varepsilon_1}$ and $h = g_n^{\delta_n} \cdots g_1^{\delta_1}$ are given by their exponent vectors $\varepsilon_1,\ldots,\varepsilon_n$ and $\delta_1,\ldots,\delta_n$ and that the routine should be able to do each of the following calculations: $h := gh$, $h := g^{-1}$, and $h := g^{-1}h$.

The special code was designed such that any collection routine simply consists of a sequence of small non-negative integers. Each instruction is represented by a leading integer characterizing its function, followed by a (possibly empty) list of parameters. It turned out that the twelve types of instructions given in table 1 suffice to formulate all required collection routines.

So the FORTRAN-written interpreter will be relatively short. An example is given in table 2. This interpreter assumes that all given buffers and lists (including the collection routine) are contained in the same large one-dimensional FORTRAN array Y.

Experience shows that the use of such an interpreter leads to quite acceptable computation times.

It should be mentioned that in our implementation we have in fact extended the list in table 1 by a few redundant instructions which represent some frequently occurring combinations of the original instructions with certain parameters. This reduces the length of the collection routine as well as the computation time at the cost of an extension of the interpreter.

4. The impact of variations

We have implemented and compared two different versions of the cm-algorithm.

The first program (version 1) follows the scheme of Jürgensen's algorithm. In particular it uses the words given in (2) to replace products of the form $g_i^r g_j^s$ with $i < j$. It turned out that this algorithm works quite satisfactorily for a lot of pc-presentations, especially for presentations of nilpotent groups with small p_i's, but that there are unfavourable examples which require immense computation times. An example for such an unfavourable pc-presentation is the following presentation of a

```
      SUBROUTINE INTERP(TYPE,E,D)
C     IF TYPE = 1, COMPUTE H = G * H.
C     IF TYPE = 2, COMPUTE H = G**(-1).
C     IF TYPE = 3, COMPUTE H = G**(-1) * H.
C     PARAMETERS ARE
C     E  = ADDRESS IN Y OF EXPONENT VECTOR EPS(1),...,EPS(N) OF G,
C     D  = ADDRESS IN Y OF EXPONENT VECTOR DELT(1),...,DELT(N) OF H,
C     LO = ADDRESS IN Y OF STACK OF COLLECTING ROUTINES,
C     P  = ADDRESS IN Y OF A LIST OF P(1),...,P(N), WHERE
C     'ADDRESS' MEANS THE INDEX OF THE LAST PRECEDING ARRAY ENTRY.
C     R IS A POINTER TO A BUFFER FOR THE RETURN JUMP ADDRESSES.
      COMMON Y(2000)/COM/N,P1,P,LO
      INTEGER TYPE,E,D,Y,P1,P,R,CODE,EPS,D1,DJ,PJ
      D1=D+1
      R=LO+3
      L=LO+Y(LO+TYPE)
      GOTO 100
    1 R=R+1
      Y(R)=L+2
    2 L=LO+Y(L+1)
  100 CODE=Y(L)
      GOTO (1,2,3,4,5,6,7,8,9,10,12,12),CODE
    3 L=LO+Y(L+2+Y(D+Y(L+1)))
      GOTO 100
    4 L=LO+Y(L+EPS)
      GOTO 100
    5 EPS=Y(E+Y(L+1))
      IF (EPS) 400,400,200
    6 EPS=Y(E+Y(L+1))
      IF (EPS.EQ.0) GOTO 400
      EPS=Y(P+Y(L+1))-EPS
  200 R=R+1
      Y(R)=L+3
  300 L=LO+Y(L+2)
      GOTO 100
    7 I=Y(L+1)
      IF (Y(E+I).EQ.0) GOTO 300
      EPS=Y(P+I)-Y(E+I)
  400 L=L+3
      GOTO 100
    8 DO 500 J=1,N
  500 Y(D+J)=0
      L=L+1
      GOTO 100
    9 Y(D1)=Y(D1)-Y(D1)/P1*P1
      RETURN
   10 Y(D1)=Y(D1)+Y(L+1)
  600 L=Y(R)
      R=R-1
      GOTO 100
   12 DJ=D+Y(L+1)
      PJ=Y(P+Y(L+1))
      Y(DJ)=Y(DJ)+Y(L+2)
      IF (Y(DJ).LT.PJ) GOTO 600
      Y(DJ)=Y(DJ)-PJ
      IF (CODE.EQ.11) GOTO 600
      L=L+3
      GOTO 100
      END
```

Table 2. Listing of an interpreter

group of order 1152 which is isomorphic to a subgroup of maximal order among the finite subgroups of $GL(4,\mathbb{Z})$:

$$M = \langle a,b,c,d,e,f,g \mid a^2=b^4=c^2=1,\ d^6=b^2,\ e^3=f^2=g^2=1,\ [a,b]=1,\ [a,c]=b^2,$$
$$[a,d]=cba,\ [a,e]=d^3,\ [a,f]=[a,g]=1,$$
$$[b,c]=b^2,\ [b,d]=cb^3a,\ [b,e]=1,$$
$$[b,f]=b^2,\ [b,g]=a,\ [c,d]=cb^2a,$$
$$[c,e]=d^3,\ [c,f]=b^3,\ [c,g]=d^3cb^3,$$
$$[d,e]=d^3b^3a,\ [d,f]=d^4cb,$$
$$[d,g]=ed^5cb^3a,\ [e,f]=e,$$
$$[e,g]=e^2d^4cb^2a,\ [f,g]=1\rangle.$$

Apparently it is the generator d which plays the crucial rôle in the behaviour of the corresponding cm-algorithm: it may occur up to the fifth power, and in fact six of the above commutators contain large powers of d. In particular this is the case for [d,e], [d,f], and [d,g] so that in many collection processes certain steps involving just one d will create several additional factors d which have to be dealt with subsequently.

To eliminate the influence of these powers of d we altered the algorithm and implemented another version (version 2) which uses the words

$$(4) \qquad \overline{g_i^r g_j^s}$$

instead of (2) to replace $g_i^r g_j^s$.

As for any such word the collection routine needs a proper subroutine, one might expect that the total length of the routine is increased remarkably by that change, but surprisingly enough, it grows only from 400 to 442. The computation time, however, is reduced drastically: Whereas (on a CYBER 76) version 1 requires $t_m = 20.15$ secs to multiply each element by its successor with respect to a lexicographical order of all group elements, $t_i = 43.70$ secs to invert

all elements, and t_c = 80.38 secs to compute the classes of conjugate elements, version 2 needs only t_m = 0.45 secs, t_i = 0.35 secs, and t_c = 2.65 secs, respectively. In table 3 also the corresponding computation times for a representation of M as group of 4×4 matrices over $\mathbb{Z}$ are given, where the figure (1152) refers to the list of all group elements which is used instead of the collection routine.

version	t_m	t_i	t_c	length
1	20.15	43.70	80.38	400
2	.45	.35	2.65	442
4×4 matrices	.48	1.43	10.81	(1152)

Table 3. Computation times for group M

Applied to a lot of test examples, version 2 turned out to be throughout faster than version 1. However, the factor between the corresponding computation times strongly depends on the presentation. In the above example it is extremely high. On the other hand, it is practically equal to 1 in case of any group of order 2^n with n generators. This is illustrated in table 4 which provides the computation times for the Sylow 2-subgroup of the Mathieu group M_{24}, given by the following presentation.

$$\begin{aligned} S = \langle a,b,c,d,e,f,g,x,y,z \mid\ & a^2=b^2=c^2=d^2=1, \\ & e^2=f^2=g^2=x^2=y^2=z^2=1,\ [a,b]=[a,c]=1, \\ & [a,d]=[a,e]=[a,f]=[a,g]=[a,x]=1, \\ & [a,y]=[a,z]=[b,c]=[b,d]=[b,e]=1, \\ & [b,f]=[b,g]=a,\ [b,x]=c,\ [b,y]=dc, \\ & [b,z]=[c,d]=1,\ [c,e]=a,\ [c,f]=1, \\ & [c,g]=[c,x]=[c,y]=[c,z]=1,\ [d,e]=a, \\ & [d,f]=1,\ [d,g]=a,\ [d,x]=[d,y]=1, \\ & [d,z]=c,\ [e,f]=[e,g]=1,\ [e,x]=f, \\ & [e,y]=1,\ [e,z]=gf,\ [f,g]=[f,x]=1, \\ & [f,y]=[f,z]=[g,x]=1,\ [g,y]=f, \\ & [g,z]=[x,y]=[x,z]=1,\ [y,z]=x \rangle. \end{aligned}$$

version	t_m	t_i	t_c	length
1	.265	.232	3.151	225
2	.258	.229	3.083	195

Table 4. Computation times for group S

Table 4 shows that the length l_2 of the collection routine constructed in version 2 may be even less than the corresponding length l_1 in version 1. For a lot of test groups of orders up to 120 the quotients l_2/l_1 varied between 0.5 and 3. The fact that for several very small groups (as in the example of table 6) the length of the collection routine exceeds by far the group order yields, of course, no problems. In general, for a fixed group presentations with more generators (and smaller indices p_i) lead to shorter collection routines than those with less generators. This may be illustrated by the cyclic group of order 729 which has been treated thrice using the following presentations.

$$Z_1 = \langle a_1,a_2 \mid a_1^{27}=1,\ a_2^{27}=a_1,\ [a_1,a_2]=1 \rangle,$$

$$Z_2 = \langle a_1,a_2,a_3 \mid a_1^{9}=1,\ a_j^{9}=a_{j-1},\ [a_i,a_j]=1 \text{ for } 1 \le i < j \le 3 \rangle,$$

$$Z_3 = \langle a_1,\ldots,a_6 \mid a_1^{3}=1,\ a_j^{3}=a_{j-1},\ [a_i,a_j]=1 \text{ for } 1 \le i < j \le 6 \rangle.$$

The results are given in table 5.

version	number of gens	t_m	t_i	t_c	length
1	2	.120	4.299	2.566	23
	3	.110	3.756	2.275	38
	6	.116	6.698	3.794	83
2	2	.060	.055	.256	256
	3	.075	.081	.305	154
	6	.111	.182	.452	148

Table 5. Computation times for cyclic group of order 729

5. The structure of the collection routine

This section is a more technical one which intends to supply those readers who might be interested in implementing their own version of the algorithm with some more detailed information on the structure of the collection routine generated in version 2.

The colletion routine consists of two main routines MULT and INV and several subroutines which, in general, are very short. MULT and INV will be called to calculate $h := gh$ or $h := g^{-1}$, respectively, where $g = g_n^{\varepsilon_n} \cdots g_1^{\varepsilon_1}$ and $h = g_n^{\delta_n} \cdots g_1^{\delta_1}$ as above. Moreover, there is an entry point INVMUL in INV which will be called in order to compute $h := g^{-1}h$. The subroutines can be subdivided into four classes. In the following I shall give a short description of all

subroutine	addr.	instruction	meaning
	1	38	address of MULT
	2	48	address of INV
	3	49	address of INVMUL
	4	0,0,0	buffer for up to n return jump addresses
AA*,A^2B	7	10,1	$\delta_1 = \delta_1+1$; return
A^2A*,AB	9	10,2	$\delta_1 = \delta_1+2$; return
AB*,AC,AC*	11	3,2,7,9	if $\delta_2 = 0$, goto AA*; if $\delta_2 = 1$, goto AB
A^2B*,A^2C,A^2C*	15	3,2,9,7	if $\delta_2 = 0$, goto A^2A*; if $\delta_2 = 1$, goto A^2B
BB*	19	11,2,1	$\delta_2 = \mathrm{mod}(\delta_2+1,p_2)$; return
BC	22	1,15	call A^2B*
	24	2,19	goto BB*
BC*,B*C*	26	3,3,19,22	if $\delta_3 = 0$, goto BB*; if $\delta_3 = 1$, goto BC
CC*,C*C*	30	12,3,1	$\delta_3 = \delta_3+1$; if $\delta_3 < p_3$, return; $\delta_3 = 0$
	33	2,15	goto A^2B*
A*C*	35	4,11,15	if $\varepsilon = 1$, goto AC*; if $\varepsilon = 2$, goto A^2C*
MULT	38	5,1,35	$\varepsilon = \varepsilon_1$; if $\varepsilon > 0$, call A*C*
	41	5,2,26	$\varepsilon = \varepsilon_2$; if $\varepsilon > 0$, call B*C*
	44	5,3,30	$\varepsilon = \varepsilon_3$; if $\varepsilon > 0$, call C*C*
	47	9	$\delta_1 = \mathrm{mod}(\delta_1,p_1)$; end
INV	48	8	$\delta_j = 0$ for $j = 1,\ldots,n$
INVMUL	49	7,3,56	if $\varepsilon_3 = 0$, goto 56; $\varepsilon = p_3-\varepsilon_3$
	52	1,11	call AC*
	54	1,30	call C*C*
	56	5,2,26	$\varepsilon = \varepsilon_2$; if $\varepsilon > 0$, call B*C*
	59	6,1,35	$\varepsilon = p_1-\varepsilon_1$; if $\varepsilon < p_1$, call A*C*
	62	9	$\delta_1 = \mathrm{mod}(\delta_1,p_1)$; end

Table 6. Collection routine for D_{12}

these routines.

Subroutines of type $G_i^r G_j^*$. For each triple (i,r,j) with $1 \le i < j \le n$ and $1 \le r < p_i$ there is a subroutine $G_i^r G_j^*$. Its task is to start the collection of the word $g_i^r g_j^{\delta_j} \cdots g_1^{\delta_1}$. Its direct effect is to check the actual value, s say, of δ_j and then to execute a jump to the according subroutine $G_i^r G_j^s$. Thus it merely consists of a kind of computed GOTO instruction.

Subroutines of type $G_i^r G_j^s$. For each quadruple (i,r,j,s) with $1 \le i < j \le n$, $1 \le r < p_i$, and $1 \le s < p_j$ there is a subroutine $G_i^r G_j^s$. It collects the word

$g_i^r g_j^s g_{j-1}^{\delta_{j-1}} \cdots g_1^{\delta_1}$ and replaces $g_j^{\delta_j} \cdots g_1^{\delta_1}$ by the result. This is done as follows. Let $\overline{g_i^r g_j^s} \equiv g_j^s g_{j-1}^{\lambda_{j-1}} \cdots g_1^{\lambda_1}$, then for each $\lambda_k \neq 0$, $1 \leq k \leq j-1$, once the subroutine $G_k^{\lambda_k} G_{j-1}^*$ is called. The last of these subroutines is called by an ordinary jump instead of a subroutine call, because there is no need to return to $G_i^r G_j^s$ after its execution.

Subroutines of type $G_i^r G_i^*$. For each pair (i,r) with $1 \leq i \leq n$ and $1 \leq r < p_i$ there is a subroutine $G_i^r G_i^*$. Its purpose is to collect the word $g_i^r g_i^{\delta_i} \cdots g_1^{\delta_1}$. It first increases δ_i by r. Then, if $\delta_i \geq p_i$, it reduces δ_i by p_i, collects the word $\overline{g_i^{p_i} g_{i-1}^{\delta_{i-1}} \cdots g_1^{\delta_1}}$ (by calling the appropriate subroutines of type $G_k^{\lambda_k} G_{i-1}^*$), and replaces $g_{i-1}^{\delta_{i-1}} \cdots g_1^{\delta_1}$ by the result.

Subroutines of type $G_i^* G_n^*$. For each i, $1 \leq i \leq n$, there is a subroutine $G_i^* G_n^*$ which is called by MULT or INV to collect the word $g_i^{\varepsilon} g_n^{\delta_n} \cdots g_1^{\delta_1}$, where ε is a global variable. It checks the actual value, r say, of ε (which is assumed to lie between 1 and $p_i - 1$) and then executes a jump to the according subroutine $G_i^r G_n^*$.

Multiplication routine MULT. The structure of MULT is very simple: For $i = 1,\ldots,n$ it sets $\varepsilon = \varepsilon_i$ and then calls subroutine $G_i^* G_n^*$, if $\varepsilon > 0$.

Inversion routine INV. After setting $\delta_j = 0$ for $j = 1,\ldots n$, INV uses the exponents $\varepsilon_n, \varepsilon_{n-1}, \ldots, \varepsilon_1$ as follows. Nothing is done, if $\varepsilon_i = 0$. Otherwise, first the word $\overline{g_i^{|g_i|-p_i} g_n^{\delta_n} \cdots g_1^{\delta_1}}$ is collected by calling the appropriate subroutines of type $G_k^{\lambda_k} G_n^*$. Then ε is set to $p_i - \varepsilon_i$, and subroutine $G_i^* G_n^*$ is called.

This long list of subroutines will be shortened remarkably by the fact that in general a lot of them are identical with other ones. Namely, if g_i^r and g_j commute, then all subroutines $G_i^r G_j^s$ with $1 \leq s < p_j$ and subroutine $G_i^r G_j^*$ can be identified with $G_i^r G_{j-1}^*$. Also, if $\overline{g_i^r g_j^s} \equiv g_j^s g_k^{\lambda_k}$ for some k, then $G_i^r G_j^s$ can be identified with $G_k^{\lambda_k} G_{j-1}^*$. Finally, if $p_i = 2$, then $G_i^* G_n^*$ can be identified with $G_i^1 G_n^*$.

Moreover, if for instance $\overline{g_i^r g_j^s} \equiv g_j^s g_{j-1}^{\lambda_{j-1}} \cdots g_k^{\lambda_k} \cdots g_1^{\lambda_1}$ and $\overline{g_i^{r'} g_j^{s'}} \equiv g_j^{s'} g_{j-1}^{\lambda_{j-1}} \cdots g_k^{\lambda_k}$ with the same exponents $\lambda_{j-1},\ldots,\lambda_k$, then subroutine $G_i^{r'} G_j^{s'}$ can be made an entry point of $G_i^r G_j^s$. In particular, if $k = 1$, it can be identified with $G_i^r G_j^s$.

The preceding remarks may be illustrated by an example. Table 6 gives the complete collection routine for the following presentation of the dihedral group of order 12:

$$D_{12} = \langle a,b,c \mid a^3=b^2=1,\ c^2=a^2,\ [a,b]=a,\ [a,c]=1,\ [b,c]=a^2 \rangle.$$

It is easy to verify the subroutines, if one computes the two additional relations which are required to apply (4), and writes the extended presentation in the form

$$D_{12} = \langle a,b,c \mid a^3=b^2=1,\ c^2=a^2,\ ab=ba^2,\ a^2b=ba,\ ac=ca,\ a^2c=ca^2,\ bc=cba^2 \rangle.$$

The list of subroutines is preceded by the addresses of MULT, INV, and INVMUL and by a buffer which is used as a push-down list to store the return jump addresses. It can easily be shown that it never needs more than n places simultaneously.

The left column shows that the theoretically given 17 subroutines of MULT and INV can be reduced to only 9 pairwise different subroutines.

References.

[1] Bayes, A. J., Kautsky, J., Wamsley, J. W.: Computation in nilpotent groups (application). pp. 82-89 in [13].

[2] Cannon, J. J.: A general purpose group theory program. pp. 204-217 in [13].

[3] Cannon, J. J.: On determining the order of a group. These proceedings.

[4] Cannon, J. J., Nicholson, T.: A draft description of the group theory language Cayley. These proceedings.

[5] Geller, E.: Ein Programm zur Bestimmung der Automorphismengruppen endlicher

auflösbarer Gruppen. Berichte der Gesellschaft für Mathematik und Datenverarbeitung 51, Bonn, 1971.

[6] Havas, G., Nicholson, T.: Collection. These proceedings.

[7] Jürgensen, H.: Calculations with the elements of a finite group given by generators and defining relations. pp. 47-57 in [8].

[8] Leech, J. (ed.): Computational problems in abstract algebra. Proceedings of a conference held at Oxford 1967. Pergamon Press, 1970.

[9] Lindenberg. W.: Über eine Darstellung von Gruppenelementen in digitalen Rechenautomaten. Numer. Math. 4 (1962), 151-153.

[10] Lindenberg, W.: Die Struktur eines Übersetzungsprogramms zur Multiplikation von Gruppenelementen in digitalen Rechenautomaten. Mitt. Rhein.-Westfäl. Inst. Instrum. Math. Bonn 2 (1963), 1-38.

[11] Macdonald, I. D.: A computer application to finite p-groups. J. Austral. Math. Soc. 17 (1974), 102-112.

[12] Neubüser, J.: Bestimmung der Untergruppenverbände endlicher p-Gruppen auf einer programmgesteuerten elektronischen Dualmaschine. Numer. Math. 3 (1961), 271-278.

[13] Newman, M. F. (ed.): Proceedings of the Second International Conference on the Theory of Groups, Canberra 1973. Lecture Notes in Mathematics, Vol. 372, Springer-Verlag, 1974.

[14] Newman, M. F.: Calculating presentations for certain kinds of quotient groups. These proceedings.

[15] Wamsley, J. W.: Computation in nilpotent groups (theory). pp. 691-700 in [13].

The Schreier Algorithm for Matrix Groups*

by

Gregory Butler

Department of Pure Mathematics
University of Sydney
Sydney 2006
New South Wales, AUSTRALIA

This paper describes an implementation of the Schreier algorithm of C.C.Sims(3), and described in J.J. Cannon(1), for matrix groups of finite dimension n over a finite field F.

Introduction

Let V be a vector space on which G acts. We will consider G as acting on the right so that the image of $v \in V$ under $g \in G$ is written v^g.

Given a set of generators for G, the algorithm computes a chain of stabilisers of subsets of V,

$$G = G^{(1)} \geq G^{(2)} \geq \ldots \geq G^{(r+1)} = 1$$

and right transversals U_i of $G^{(i+1)}$ in $G^{(i)}$ for $i = 1,2,\ldots,r$.

Using the notation that $G_X = \{g \in G \mid X^g = X\}$ is the setwise stabiliser in G of a subset X of V and inductively defining $G_{X_1,X_2,\ldots,X_i} = (G_{X_1,X_2,\ldots,X_{i-1}})_{X_i}$ we can extend the definition of C.C. Sims(4) and define a base for G to be a sequence $B = X_1,X_2,\ldots,X_r$ of subsets of V such that $G_{X_1,X_2,\ldots,X_r} = 1$.

The algorithm computes a base $B=X_1,X_2,\ldots,X_r$ for G, generators for $G^{(i)} = G_{X_1,X_2,\ldots,X_{i-1}}$ for $i = 2,3,\ldots,r$, and the right transversals $U_1,U_2,\ldots,U_r$.

This allows us to

1) determine the order of G,
2) determine if an element of GL(n,F) is in G, and
3) step through the elements of G without repetition.

We can do this because each element g of G has a unique representation $g = u_r u_{r-1} \ldots u_1$ with $u_i \in U_i$ for each i.

If g is an element of GL(n,F) and X_1^g is not in the G-orbit of X_1 then g is not an element of G. On the other hand, if there is a $u_1 \in U_1$ such that $X_1^g = X_1^{u_1}$ then gu_1^{-1} fixes X_1 and we need to determine if gu_1^{-1} is an element of G_{X_1}. Continuing in the same way we either determine that g is not an element of G or we find $u_1,u_2,\ldots,u_r$ with $u_i \in U_i$ such that $gu_1^{-1}u_2^{-1}\ldots u_r^{-1}$ fixes $X_1,X_2,\ldots,X_r$. Then g is an element of G if and only if $g = u_r u_{r-1} \ldots u_1$.

Theory

The algorithm depends on two results. The first applies to all finite groups, while the second is a generalization of a result for permutation groups.

Schreier's Lemma (M.Hall(2,p96))

Let G be a finite group generated by a set S. Let H be a subgroup of G and let U be a right

* This work was supported in part by the Australian Research Grants Committee.

transversalfor H in G. Then H is generated by $\{us\phi(us)^{-1} | u\varepsilon U, s\varepsilon S\}$, where $\phi(g)$ is the unique element of U such that Hg = Hu.

Lemma 2

Let X be a subset of V and let $g,h \varepsilon G$. Then g and h belong to the same right coset of G_X if and only if $X^g = X^h$.

Starting at level 1 the general procedure is:

At each level i we form all the products us with $u \varepsilon U_i$ and s a generator for $G^{(i)}$. We determine if us is in a coset of $G^{(i+1)}$ already found by comparing X_i^{us} and $X_i^{u_i}$ for all $u_i \varepsilon U_i$. If it is not then us is added to U_i.

If us is in the coset that is already represented by u_i we form usu_i^{-1} and see if it can be expressed in the form $u_r u_{r-1} \ldots u_{i+1}$ with $u_j \varepsilon U_j$. If it can we form the next product at level i. If it cannot we add usu_i^{-1} to the generators for $G^{(i+1)}$ and continue at level i+1.

We return to level i-1 when we have formed all the products us at level i.

We are finished when we have finished at level 1.

Implementation

The main difference between the implementation of the Shreier algorithm for permutation groups and matrix groups is in the choice of base elements and comparison of images of the base elements under the action of elements of the group.

The choice of base elements is critical as we wish to factor the order of $G - |G| = \prod_{i=1}^{r} |U_i|$ - so that none of the terms $|U_i|$ is too large. Not only must we store each element of each transversal, but large $|U_i|$ mean that the number of products us that must be formed is large.

The choice is also critical as the test $X_i^{us} = X_i^{u_i}$ must be efficient. For permutation groups this test is just a comparion of two points. For matrix groups it is the comparison of two sets of vectors. Thus the problem is to make this comparison as efficient as possible.

The approach we took was to choose $X_{2i-1} = \langle v_i \rangle = \{\lambda v_i | \lambda \varepsilon F\}$ and $X_{2i} = \{v_i\}$, for some $v_i \varepsilon V$. That is, we first stabilise a one dimensional subspace and then stabilise a vector in this subspace.

For each one dimensional subspace X of V we define a canonical representative $\underline{x}$ as follows: $\underline{x}$ is the unique vector of X whose first non-zero entry is the identity of F, when the elements of V are considered as row vectors. Since G acts linearly on V, the image of $\langle v_i \rangle$ under $g \varepsilon G$ is $\langle v_i^g \rangle$ and so our test reduces to a comparison of the canonical representatives of the subspaces $\langle v_i^{us} \rangle$ and $\langle v_i^{u_i} \rangle$.

Likewise, when the base element is $\{v_i\}$, the test is just a comparison of two vectors, v_i^{us} and $v_i^{u_i}$.

Thus in both cases the test is a comparison of two vectors. In fact the images $v_i^{u_i}$ of the cosets are stored and we compare v_i^{us} with each $v_i^{u_i}$ by doing a 'hash' search of these images. (Sz(32) took 101 seconds when we used a sequential search whereas it took 85 seconds using a 'hash' search.)

This gives an efficient method of doing the test but it is still not known exactly how well this choice of base will factor the order of G.

Results

We give some results obtained in Sydney on a CDC Cyber 72 in Table 1. Here the choice of v_i for the base is the vector with the identity in the i-th position and zero elsewhere.

TABLE 1.

GROUP	DEGREE n	ORDER	F=GF(q) q	TIME seconds	$\|U_i\|$ i=1,2,...,r
SL(2,9)	2	720	9	0.35	10,8,9
SL(2,31)	2	29,760	31	1.51	32,30,31
$U_3(3)$	3	6,048	9	1.78	28,8,9,1,3
$Sp_4(2)$	4	720	2	1.45	15,1,8,1,3,1,2
$Sp_4(3)$	4	51,840	3	7.59	40,2,27,1,4,2,3
$Sp_4(5)$	4	9,360,000	5	37.79	156,4,125,1,6,4,5
Sz(8)	4	29,120	8	3.81	65,7,8,1,8
Sz(32)	4	32,537,600	32	85.40	1025,31,32,1,32
$U_5(2)$	5	13,685,760	2	28.77	176,3,40,3,12,3,2,3
$U_3(3)xSL_29$	5	4,354,560	9	9.58	28,8,9,1,3,1,10,8,9
J_2	6	604,800	4	132.96	525,3,96,1,4
$Sp_6(2)$	6	1,451,520	2	22.11	63,1,32,1,15,1,8,1,3,1,2
$Sp_6(3)$	6	9,170,703,360	3	154.04	364,2,120,2,36,2,27,1,9,1,3
$Sp_8(2)$	8	47,377,612,800	2	308.72	255,1,128,1,63,1,32,1,15,1,8,1,3,1,2
$Sp_{10}(2)$	10	24,815,256,521, 932,800	2	3513.59	1023,1,512,1,255,1,128,1,63,1,32,1, 15,1,8,1,3,1,2

Analysis

For each i let $n_i = |U_i|$ and let s_i be the number of generators for $G^{(i)}$. Let $d = \sum_{i=1}^{r} n_i$ and $e = \sum_{i=1}^{r} s_i$. Then the total storage required for the coset representatives, generators, and images of the cosets is

$$dn^2\log_2|F| + en^2\log_2|F| + dn\log_2|F|$$

because we require a field of $\log_2|F|$ bits to store an element of F.

For permutation groups of degree n the similar equation is

$$dn\log_2 n + en\log_2 n + rn\log_2 n$$

The time consuming task of the algorithm is forming the $\sum_{i=1}^{r} n_i s_i$ products of the form us. For matrices a product takes n^3 operations while for permutations only n operations are required.

The time necessary for forming the images X_i^{us} in the two cases is similarly related. The process takes n^2 operations in the matrix case and one operation in the permutation case.

Thus the total time is proportional to
$(n^3 + n^2) \sum_{i=1}^{r} n_i s_i$ for matrix groups and
$(n + 1) \sum_{i=1}^{r} n_i s_i$ for permutation groups.

These differences in storage and time taken are compensated for by the fact that a matrix representation of a group often has much lower degree than a permutation representation. For example, $U_3(3)$ has a permutation representation of degree 28. The permutation Schreier algorithm takes 2.11 seconds and factors the order as 28x27x8. The matrix representation has degree 3 over GF(9), takes 1.78 seconds and the order factors as 28x8x9x1x3.

Conclusion

No claim is made that either the algorithm or the implementation is the best possible. Indeed variations to both have been suggested and the merit of each is open to study.

The storage of each coset representative can be avoided by expressing each representative as a word in the generators. This can greatly reduce the

storage requirements but, as each coset representative must be multiplied out whenever it is needed, it increases the time taken.

The algorithm has been implemented for matrices over the ring of integers and its implementation for matrices over other fields and rings is possible.

A large proportion of the time taken by the algorithm is spent checking that all the cosets have been formed. To avoid this it might be possible to use a Todd-Coxeter coset enumeration to do the checking and also to suggest which products us will be most useful in giving new cosets.

Acknowledgements

I would like to take this opportunity to sincerely thank Dr. J.J. Cannon for his continuing guidance and encouragement. I would also like to thank Mr. P.T.J. Nicholson for his help and his many suggestions.

Bibliography

1. J.J. Cannon - On determining the order of a group, in these proceedings.

2. M. Hall, Jr. - The Theory of Groups, MacMillan, New York, 1959.

3. C.C. Sims - Computational methods in the study of permutation groups, in Computational Problems in Abstract Algebra, J. Leech (editor), Pergamon Press, Oxford and New York, 1970.

4. C.C. Sims - Determining the conjugacy classes of a permutation group, in SIAM-AMS Proceedings, vol. 4, AMS, Providence, R.I., 1971.

CAMAC

by

Vera Pless

University of Illinois

Chicago, Illinois

CAMAC (Combinatorial and Algebraic Machine Aided Computation) is a computer system to do algebraic and combinatorial computations in a unified manner. The CAMAC system [13, 16] is to interface with the GROUP system [1,2,3,4,5,9,10,11] of John Cannon and J. Neubüser to extend its facilities to new fields of study.

As an example of how computers have interacted with mathematics in the past, one can look at the recent history of the sporadic groups. Finite group theory has become very exciting recently, in part because computers could be used to do very complicated computations. Another reason this subject is exciting is the recent discovery of new, simple sporadic groups. In order to describe this I will give you a very brief history of the simple sporadic groups. All simple groups except for a few exceptional ones, the sporadic simple groups, belong to infinite families of groups. About 100 years ago, Mathieu (1861) discovered five groups of this type, the first sporadic groups. No new sporadic groups were discovered until 1966 when Janko, the Jugoslav mathematician, discovered a new simple, sporadic group and since then two or three new ones have been found every year. Computers have entered this area in many different ways. The Todd-Coxeter algorithm has been used many times in investigating properties of these new groups. This algorithm, a computational way of determining the index of a subgroup in a group given by generators and relations, was devised before the advent of computers. The Todd-Coxeter algorithm has since been modified, in several different fashions, so that it can now run on a computer and it is the main algorithm to handle these types of groups [8]. Another area in which computers have proved useful is in the construction of character tables for these new groups [7]. A major problem in the study of the new simple sporadic groups is the determination of the existence of each candidate, and computers have been very vital in this regard. In many cases the existence of these groups has been demonstrated because they have been shown to be the automorphism groups of combinatorial structures. C.C. Sims has done this type of thing and in the process has developed efficient methods to handle large permutation groups, as the new sporadic groups are indeed very large [15]. Most of these computations were done with special purpose programs; i.e., programs written for the express purpose of doing a specific computation and usually never used again. Many of the techniques developed for such programs can be turned into algorithms to be used over and over again in a unified system such as CAMAC.

The first system of programs organized together to handle groups in a unified manner was established by Joachim Neubüser in Kiel, Germany about 15 years ago [9]. This group is presently located in Aachen, Germany. In 1965, John Cannon [1] at Sydney independently started writing a system of programs for group theory computations. Both the original Kiel and Sydney programs were written in assembly code. Another system of this type is GRAPPA at Bell Laboratories which was established by John Cannon and Lou Dimino but which is not undergoing development at present. In 1972, Neubüser and Cannon commenced a joint project to write a new group theory system (called GROUP). This system was to contain many new algorithms and to be as machine independent as possible. On this continent, about two and a half years ago, we have implemented an early version of the GROUP system and started building the CAMAC system compatibly with GROUP such that it can utilize the power of the GROUP system, when appropriate, in the solution of particular problems. The CAMAC system is, in a sense, extending the GROUP system in other directions. The beginning of CAMAC was made at M.I.T. and involved the translation of programs written in Fortran and assembly language for the CDC-6600 to programs capable of running on a PDP-10 at M.I.T. When I moved to Chicago Circle, the entire system, which was twice the size of the original one, had to be again converted to run on the Circle Computation Center IBM 370-158 computer. This large conversion is being ably done by William Patton of our group. It is planned that also in the future the CAMAC system will be interfaced with the further additions to the GROUP system.

The two main algorithms for handling finite groups on computers are the Todd-Coxeter algorithm [8] and the Sims way of handling permutation groups via a chain of stabilizer subgroups [15]. Both of these approaches have been incorporated into related algorithms that perform new operations. For instance, the Todd-Coxeter algorithm has appeared in programs to find defining relations for a permutation group [3], and in the index program [6] which finds a subgroup of a given index in a group given by generators and relations. The implementations of the Todd-Coxeter algorithm and for

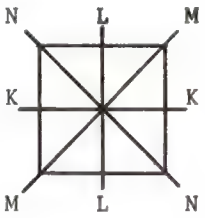
Supported by NSF Grant MCS 76-03143A01

finding relations for a permutation group are part of the GROUP system. These types of advances plus the usefulness of these programs in determining specific characteristics of groups have led to a new area of endeavor, namely, the development of efficient computer algorithms for algebraic and combinatorial computations and their implementation in a unified system like CAMAC. In addition to the purely group theoretic algorithms described above, I have been interested in permutation groups acting on combinatorial structures. One of the algorithms which I developed in this direction, with my students, is a vector orbits algorithm which determines the orbits of binary vectors acted upon by a permutation group [13]. I used this to solve a problem in "Code-Graphs." In addition to this coding problem, CAMAC played a major role in analyzing other self-dual codes over the field of 3 elements [14]. This work uses the vector orbits algorithm, set stabilizer algorithm, and a code weight distribution plus other CAMAC facilities. In this paper, the complete weight enumerators of self-dual codes over GF(3) are studied. In addition to general results, all maximal self-orthogonal codes of length ≤ 12 and specific self-dual codes of lengths 24, $\overline{36}$, 48, and 60 were studied.

The CAMAC system is to be of aid to people doing research in group theory or combinatorial theory and other scientists who use groups such as crystallographers and people working with error correcting codes. At times it will be an extension of paper and pencil. I have encountered many times situations in which I had permutations generating a group but did not know anything about the group. The first thing I want to know is the order of the group generated by these permutations. Our aim is to handle very large groups or combinatorial configurations so we need clever algorithms as brute force methods will not work. I have also been confronted with groups given by generating matrices. Again it was important to know the order of this group. In general, the system is versatile enough to test various premises about the group under consideration. What is its order, order of subgroups, conjugacy classes, etc. Hopefully, the system would one day be powerful enough to establish the existence of new, finite simple groups. The new Fischer group has more than 10^{24} elements, and so far has been very difficult to handle by computer or in any other way. However, even before CAMAC reaches this point, it is quite valuable as a research aid.

Now I will describe the system more in detail. Using the presently interfaced parts of the GROUP system the CAMAC system accepts groups given by either generating permutations or generators and relations. We illustrate this with the following example.

Example: Group with 8 elements

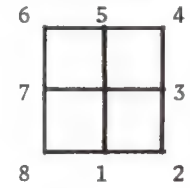


Motions of the Square

I: Identity
a: Rotate 90° counter-clockwise
b: Rotate 180° counter-clockwise
c: Rotate 270° counter-clockwise
d: Flip over K-K axis
e: Flip over L-L axis
f: Flip over M-M axis
g: Flip over N-N axis

If we repeat these operations one after the other, we get another operation in the set. Also, for every operation there is another operation which repeated after it is the identity. Hence this is a group.

There are many ways this group can be described. One way is by a multiplication table. For this group, the table would have $8^2 = 64$ entries. Another way is by generators and relations; i.e., $a^4 = d^2 = 1$, $dad = a^3$. Still another way is by generating permutations.



Here we label the vertices with numbers.

(1,3,5,7)(2,4,6,8) and
(2,4)(1,5)(6,8)

These two ways of describing groups do not take much space; further we have a very large group determined by a few generating permutations or a small number of generators and relations, whereas a multiplication table for a group of order n must have n^2 entries. Further, these types of descriptions occur naturally when people are working with groups. However, with this description we do not know much about the group, not even its order.

When a group is described as above to CAMAC, it is given a name. Then, using the presently interfaced parts of the GROUP system, a table is set up in the name of the group, and everything computed about the group is stored in this table. We can, at any time we like, print out any item in the table or the whole table. We can also give certain combinatorial structures, like error correcting codes, to CAMAC.

The CAMAC system works either interactively or on batch. Generally, we use the former mode for small problems and the latter mode for large problems. It is written mainly in Fortran with parts in assembly language, but the user does not need to know either language to use the system. We have a user interface. This consists of commands written in simple English which are contained in a manual [16] which is available for the asking. It is quite easy to learn how to use the system, and whenever one forgets what to do next it is possible to query the system and receive an answer. We illustrate some of these points in the following.

Sample Output

Command: Read
Type: P
Name of New Group: PGL(5)
Degree: 6
Options: N
Permutations (cycle): (2,3,4,5,6),(1,2)(3,6),
(3,4,6,5)
Command: Generate Group
Name of Group:
Command: Class
Options: Y
Save Classes?: Y
Command: Relate
Command: Print
Item: -1
Group PGL(5)
Presentation

B^2, CBCB, C^{-4}, A^5, A^2 CAC^{-1}, $BA^{-1}BA^{-1}BA^{-1}$

Print Perms Image Form (Default = Cycle)

1. (2,3,4,5,6)
2. (1,2)(3,6)
3. (3,4,5,6)

Number of Generators: 3
Order of Group: 120
Class Table:

1	1	1
2	5	24
3	2	15
4	3	20
5	4	30
6	2	10
7	6	20

The above represents some sample output of CAMAC on the Circle Center time sharing system. This format is from the Aachen-Sydney Group System. The words which are not underlined are generated by CAMAC and a colon indicates that a response from the user is required. The underlined words are the user's response. After certain responses, CAMAC requests the name of the group. A default refers to the last group named. This question is only listed after GENERATE GROUP above although it occurs in actual usage after the commands CLASS, RELATE, and PRINT. The command GENERATE GROUP computes the order of PGL(5), CLASS computes the conjugacy classes, RELATE computes defining relations and PRINT, followed by -1 in response to ITEM, prints out all the information about PGL(5) which has been generated by these commands.

The block diagram of CAMAC illustrates the organization of CAMAC. In this diagram a group or combinatorial structure is input into CAMAC either on batch or interactively. The programs to handle a group given by defining relations are based on the Todd-Coxeter algorithm and many of these are listed on the left. Many of the algorithms to handle groups given by generating permutations are listed in the center section. The outputs of both of these sections enter the group state table. Some of the algorithms to handle combinatorial structures are given on the right. This block diagram shows how we have extended the Aachen-Sydney GROUP system with CAMAC capabilities.

The following examples illustrate some representative problems which CAMAC has already solved.

1. Sp (4,4) is a permutation group of degree 85 and order 979,200. It is a transitive group. The stabilizer of a point has 3 orbits of sizes 1, 20, and 64. This subgroup has order 11,520.

   Is the stabilizer subgroup faithful on the orbits of size 20 and size 64?

2. J_2 is a group of order 604,800 given by the following relations:

   $a^3=b^3=c^3=(ac)^5=(bc)^5=(baba^{-1}b^{-1}a^{-1}) =$

   $(ac^{-1}ac)^2=(bc^{-1}bc)^2=$

   $(a^{-1}baca^{-1}bac^{-1}a^{-1}b^{-1}ac^{-1}) =$

   $(b^{-1}abcb^{-1}abc^{-1}b^{-1}a^{-1}bc^{-1}) = 1$

   What are the generators for the centralizer of the involution $ca^{-1}ca$ in J_2 ?

3. The (36,18) symmetry code over GF(3) has 888 vectors of weight 36. This code has a group G of monomial transformations acting on it of order 19,584.

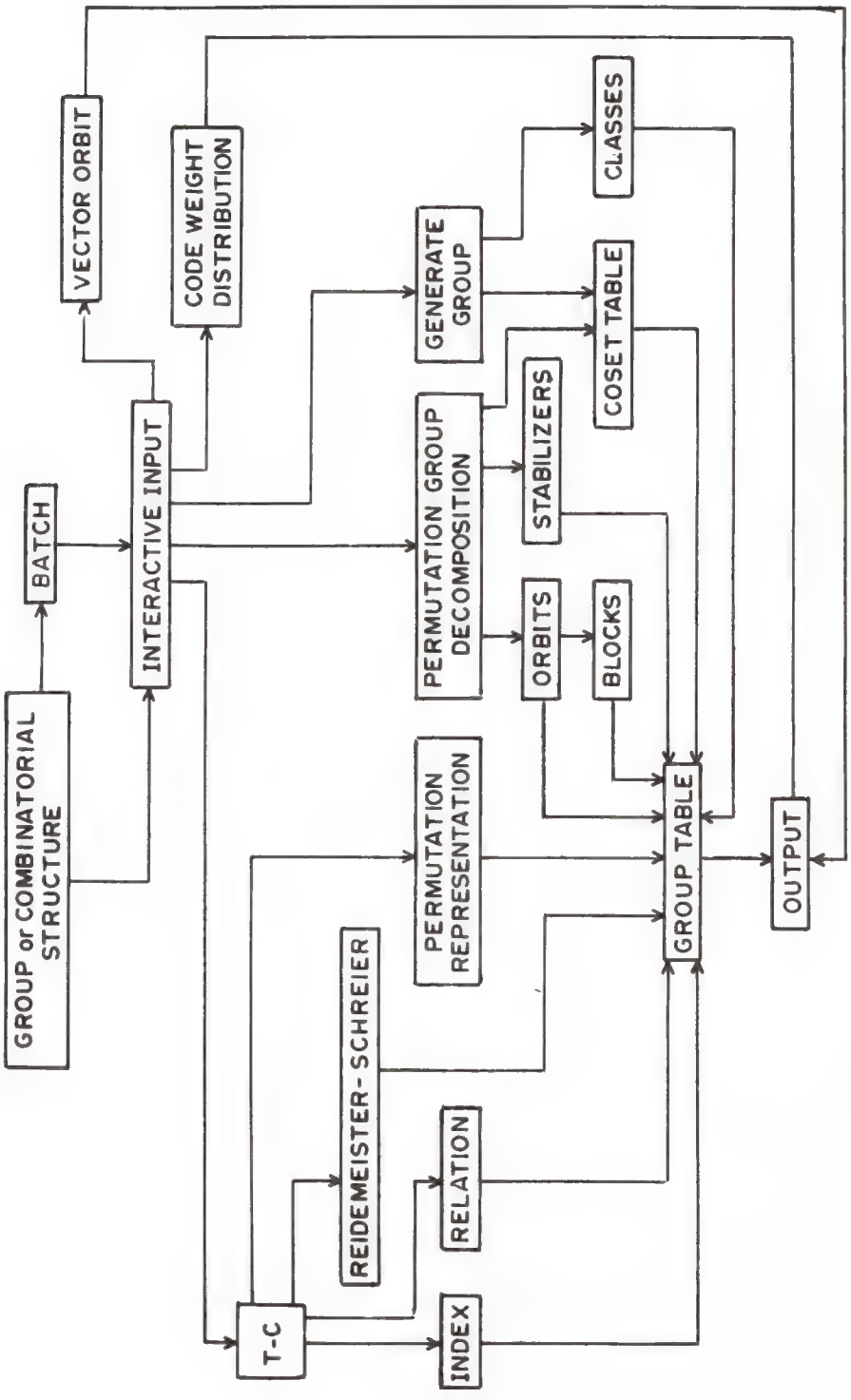
   How large are the orbits of the maximal weight vectors under the action of G ?

For the first problem, posed by John McKay, GROUP programs available via CAMAC determined that the stabilizer subgroup is not faithful on the orbit of size 20 and is faithful on the orbit of size 64.

The second problem also arose in the work of John McKay. Using a modification of an implementation of the Todd-Coxeter algorithm from the GROUP system, CAMAC was able to help him solve this problem by printing out coset representatives of J_2 over a certain subgroup of index 840.

The third problem was part of my work on determining the complete weight distributions of this code. There is one orbit of size 72 and one of size 816. Using the CAMAC vector orbits program the orbits of the 41,184 maximal weight vectors in the (60,30) symmetry code under the group G known to act on the code are as follows:

Block Diagram of CAMAC



6. Improve user interface to make the system easier to use and able to handle more of the users needs.

7. Improve the system so that it's easier to debug and to add capabilities.

8. We want to develop problems for CAMAC so that it can be used to aid in teaching group theory to students.

I wish to thank the Circle Computation Center for all the aid they have given me in the work of implementing CAMAC on their IBM 370-158.

BIBLIOGRAPHY

1. John Cannon, "Computing Local Structure of Large Finite Groups," Computers in Algebra and Number Theory, Vol. IV, SIAM-A.M.S. Proceedings, Providence, Rhode Island, 1971.

2. John Cannon and James Richardson, "The GROUP System for Investigating the Structure of Groups, User and Maintenance Manual," Technical Report No. 8, Department of Pure Mathematics, The University of Sydney, 1973.

3. John Cannon, "Construction of Defining Relators for Finite Groups," Discrete Math., V. 5, 1973, pp. 105-129.

4. John Cannon, Robyn Gallagher, and Kim McAllister, "Stackhandler: Language Extension for Low Level Set Processing, Programming and Implementation Manual," Technical Report No. 5, Department of Pure Mathematics, The University of Sydney, 1974.

5. John Cannon, "A General Purpose Group Theory Program," Proc. 2nd Internat. Conf. Theory of Groups, Canberra 1973, Lecture Notes in Mathematics, Vol. 372, Springer-Verlag, Berlin, 1974, pp. 204-217.

6. A. Dietze and M. Schaps, "Determining subgroups of a given finite index in a finitely presented group," Canad. J. Math. 4 (1974), pp. 769-782.

7. M. Hall, Jr., "Computers in Group Theory," Summer School on Group Theory and Computation, Galway, 1973.

8. J. Leech, "Coset Enumeration," Computational Problems in Abstract Algebra, Pergamon Press, Oxford, 1970, pp. 21-35.

9. J. Neubüser, "Untersuchungen des Untergruppenverbandes endlicher Gruppen auf einer programm--gesteurten elektronischen Dualsmaschine," Numerische Mathematik, Vol. 2, 19 0, pp. 280-292.

10. J. Neubüser, "Investigations of Groups on Computers," Computational Problems in Abstract Algebra, Pergamon Press, Oxford and New York, 1970, pp. 1-20.

11. J. Neubüser, "Some Computational Methods in Group Theory," 3rd Internat. Colloquium on Advanced Computing Methods in Theoretical Physics, Centre de Physique Theorique, C.N.R.S. Marseille, 1973, pp. B-11-1-B-11-35.

12. Vera Pless, "Symmetry Codes and Their Invariant Subcodes," Journal of Combinatorial Theory (A), V. 18, 1975.

13. Vera Pless, "CAMAC," Proceedings of the 6th Southeastern Conference on Combinatorics, Graph Theory, and Computing, Florida Atlantic University, Boca Raton, Florida, February 1975, pp. 505-511.

14. Vera Pless and N.J.A. Sloane, "Self-Dual Codes Over GF(3)," to appear in SIAM J. on Applied Math., Part B.

15. C.C. Sims, "Computation with Permutation Groups," Second Symposium on Symbolic and Algebraic Manipulation Proceedings (Assoc. for Computing Machinery, 1971).

16. Randell Weiss, "How to Use the CAMAC Group Manipulation System," Project MAC Technical Memo 16, 1975.

17. Randell Weiss, "Finding Isomorph Classes for Combinatorial Structures," Project MAC Technical Memo 64, 1975.

Orbit size	Order of the stabilizer in G of a vector in the orbit	Comments
120	812	60 of these vectors form a Hadamard matrix and the other 60 are their negatives.
9,744	10	They project onto maximum weight vectors in Golay (12,6) codes and are described in [12].
6,960	14	Also related to Golay (12,6) codes.
24,360	4	
41,184:Total		

The following lists some of the capabilities available at the present time in CAMAC itself or in GROUP via CAMAC.

Brief Description of CAMAC

Types of Programs Now Available

1. For permutation group:

   Order, coset table over a subgroup, stabilizer subgroups, centralizer subgroups, normalizer subgroups, conjugacy classes, orbits, blocks, generators and relations, etc.

2. For groups given by generators and relations:

   Index of a subgroup, coset representatives, relations for a subgroup, permutation representation.

3. For a permutation group considered as acting on vectors we have vector orbits.

The main tool for handling permutation groups is the Sims way of storing it by coset representatives of a chain of stabilizer subgroups [15]. By means of this, information about very large groups can be computed efficiently. The algorithm for computing defining relations is due to John Cannon [3]. The implementations are part of the GROUP system.

The main algorithm for groups given by generators and relations is the Todd-Coxeter algorithm. The implementation is part of the GROUP system.

CAMAC is the only system mentioned above with the vector orbits capability. This is described in Weiss [17].

The uses of computers in the investigations of groups and combinatorial structures opens up many new areas of research. Our primary aim is to develop efficient computer algorithms to perform this type of computation. New algorithms are needed and many of the old algorithms now working could be improved or even replaced by new ones which could handle a wider variety of computations. We want to be able to handle very large groups or combinatorial configurations. We need clever algorithms which will work quickly and make efficient use of space. Also there are possibilities of making ingenious variations of existing algorithms to perform tasks which they do not now perform. An example of this type of thing is the Todd-Coxeter algorithm which computes the index of a subgroup in a group given by generators and relations when the subgroup generators are specified in terms of the generators of the entire group. A modification of this algorithm produces the INDEX program [6] which has four options. One is the determination of any subgroup of index K (for moderate sized K); another is the determination of all subgroups of index K , and the third and fourth options are like the first and second with K replaced by $<K$. For moderate sized K this algorithm can determine the order of the main group at times when the Todd-Coxeter cannot. This latter happens when the Todd-Coxeter program does not terminate but the index program does. This type of phenomenon is not unusual, i.e., where one algorithm is modified to produce another one.

One of the main problems in this area is to determine which of several algorithms is the most efficient or circumstances under which one algorithm is more efficient than another.

The following are some of our future plans:

1. Strengthen present capabilities.

2. Add programs on group characters.

3. Add the capability of handling groups given by generating matrices.

4. Expansion in the direction of handling combinatorial objects; particularly

   a) coding package, and

   b) groups of combinatorial configurations.

5. Both for handling combinatorial configurations and permutation groups of large degree we need the following capabilities:

   a) subgroup stabilizing a vector,

   b) intersection of subgroups.

A Program for Determining Fundamental Units

by Michael Pohst, Mathematisches Institut der Universität zu Köln

§1 Introduction

Determining a system of fundamental units in algebraic number fields is a very old problem. It naturally arises, if we want to solve Diophantine equations, compute class numbers, etc.. The structure of the group of units is given by Dirichlet's Unit Theorem. Unfortunately all proofs only state that a system of fundamental units can be found in a finite number of steps. But there are no bounds for this number.

Nonetheless it is possible to refine the theory so that all computations can be carried out by machine. Recently I wrote a computer program in Fortran which will determine a full system of fundamental units for a given algebraic number field (under certain premises concerning the available CPU-time and storage). I shall describe and discuss this program in the following. The theory we shall use was developed by H. Zassenhaus and myself in [2], [4]. Naturally the program is written to work in all cases. If we deal with any special field - with known subfields for example, we can use the characteristic features to make the program still more efficient.

§2 Preliminary considerations

Our aim is to determine a system of fundamental units of an algebraic number field F of degree $n \geq 2$. Let F be given by $F = \mathbb{Q}(\beta)$ with a root β of the monic irreducible polynomial

(1) $f(x)=x^n+a_1x^{n-1}+\ldots+a_{n-1}x+a_n$

($a_i \in \mathbb{Z}$, $i=1,\ldots,n$). We shall assume F totally real, i.e. all zeros of f must be real. The program can also handle the complex case after some modifications.

Let o_F denote the ring of integers of F. To represent every element of o_F as a vector $(b_1,\ldots,b_n) \in \mathbb{Z}^n$ we need a $\mathbb{Z}$-basis of o_F which can be determined by machine from (1) by methods of Nagell, Zassenhaus ([1], [6]). I shall not go into this, since in most cases there is a generating polynomial (1) so that the successive powers $1,\beta,\ldots,\beta^{n-1}$ form an integral basis of F. Therefore we shall actually determine a system of fundamental units of the order $o_\beta := \mathbb{Z}+\beta\mathbb{Z}+\ldots+\beta^{n-1}\mathbb{Z}$. Let $o_\beta = o_F$ from now on. For $o_\beta \subsetneq o_F$ a change to an integral basis must be made. Every number $z \in o_\beta$ is of the form

(2) $z=b_1+b_2\beta+\ldots+b_n\beta^{n-1}$, $(b_1,\ldots,b_n) \in \mathbb{Z}^n$.

We can add (subtract) two numbers $y,z \in o_\beta$ by adding (subtracting) their components. Also multiplication and division (if $\frac{y}{z} \in o_\beta$) can be done by integer arithmetic, if higher powers β^m ($m \geq n$) are reduced by (1). Remember β is a root of f.

But when we divide, we must already solve a system of linear equations, and we would prefer floating point arithmetic. This will be of advantage also for the computations of norms and traces. So we regard F as a subfield of the real numbers $\mathbb{R}$. The polynomial f has n distinct roots

$\beta^{(1)}=\beta,\ldots,\beta^{(n)}$. We compute them by the method of Newton-Maehly [5] in double precision. The distinct roots correspond to the conjugate fields of F: $F^{(1)}=F,\ldots,F^{(n)}$. We store the values $(\beta^{(i)})^j$ $(i=1,\ldots,n;$ $j=1,\ldots,n-1)$. For numbers $z\in o_\beta$ given by (2) we easily compute the values $z^{(i)}$ approximately and then get the rational integer $N(z)$ $(Tr(z))$ as the nearest integer to the product $z^{(1)}\cdot\ldots\cdot z^{(n)}$ (sum $z^{(1)}+\ldots+z^{(n)}$). This is important, since $z\in o_\beta$ is a unit if and only if $N(z)=\pm 1$.

As mentioned above we will also solve systems of linear equations by floating point arithmetic. The desired solutions are again found by taking the nearest integers. Of course we must be cautious, if the corresponding matrix is ill conditioned. For example we determine a dual basis $\bar\beta_1,\ldots,\bar\beta_n$ (together with their conjugates) by solving the systems of linear equations

$$\begin{array}{rcl} \bar\beta_i^{(1)}+\ldots+\bar\beta_i^{(n)} &=& \delta_{i1} \\ \beta^{(1)}\bar\beta_i^{(1)}+\ldots+\beta^{(n)}\bar\beta_i^{(n)} &=& \delta_{i2} \\ \vdots \qquad\qquad \vdots && (i=1,\ldots,n). \\ (\beta^{(1)})^{n-1}\bar\beta_i^{(1)}+\ldots+(\beta^{(n)})^{n-1}\bar\beta_i^{(n)} &=& \delta_{in} \end{array}$$

In the totally real case by Dirichlet's Theorem the group E of all units of o_β is the direct product of n-1 infinite cyclic groups and ± 1: $E=\langle -1\rangle\times\langle\varepsilon_1\rangle\times\ldots\times\langle\varepsilon_{n-1}\rangle$. A set of elements $\{\varepsilon_1,\ldots,\varepsilon_{n-1}\}$ which generates $E/\langle -1\rangle$ is called a system of fundamental units. Let $m\in\mathbb{N}$ and $\eta_1,\ldots,\eta_m\in E$. We say that $\eta_1,\ldots,\eta_m$ are independent, if there is no solution $(k_1,\ldots,k_m)\in\mathbb{Z}^m$ of the equation

(3) $1=\eta_1^{k_1}\cdot\ldots\cdot\eta_m^{k_m}$

other than $(0,\ldots,0)$. For n-1 independent units $\eta_1,\ldots,\eta_{n-1}$ the determinants of all submatrices of n-1 rows of the matrix

$$\begin{pmatrix} \log|\eta_1^{(1)}| & \ldots & \log|\eta_{n-1}^{(1)}| \\ \vdots & & \vdots \\ \log|\eta_1^{(n)}| & \ldots & \log|\eta_{n-1}^{(n)}| \end{pmatrix}$$

have the same absolute value, it is called regulator R_η of $\eta_1,\ldots,\eta_{n-1}$. All systems of fundamental units have the same regulator R of the field F. Then $\varepsilon_1,\ldots,\varepsilon_{n-1}\in E$ form a system of fundamental units of F, if and only if they are independent and their regulator is minimal. Thus the problem of determining a system of fundamental units splits into two steps. First we compute independent units $\eta_1,\ldots,\eta_{n-1}$ in §3. In §4 we then describe two methods to modify $\eta_1,\ldots,\eta_{n-1}$ so that we again get n-1 independent units but with smaller regulator. In this way we change $\eta_1,\ldots,\eta_{n-1}$ gradually into a system of fundamental units. A useful tool for both paragraphs is the fact that the index of the group G_η generated by -1 and n-1 independent units $\eta_1,\ldots,\eta_{n-1}$ in the full unit group E is equal to the quotient $\frac{R_\eta}{R}$. We obtain an estimate for this index from above, if we know a lower bound for R and compute R_η. A bound for R is given by

(4) $R > \sqrt{\frac{1}{n}\left(\frac{0.231555\ n}{2}\right)^{n-1}}\ \frac{1}{\Gamma(\frac{n+3}{2})}$.

Inequality (4) can be slightly improved for $n\leq 9$. There exist other bounds which also depend on the field discriminant d_F [3].

§3 Computation of independent units

Our aim is to construct as many elements of o_β of bounded norm as we like. Their existence is guaranteed by Minkowski's Convex Body Theorem. Every such element of norm ± 1 is already a unit, and quotients of elements of the same norm are units, if they belong to o_β. It can be shown that we at last get n-1 independent units in this way [4].

To apply Minkowski's Theorem we define a mapping $\varphi : o_\beta \to \mathbb{R}^n$:
$(b_1+b_2\beta+\ldots+b_n\beta^{n-1}) \to (b_1,\ldots,b_n)$.
Clearly φ is an additive monomorphism of o_β onto the n-dimensional lattice $\mathbb{Z}^n$ with mesh 1. Let us consider the parallelotope
$P := \{(c_1,\ldots,c_n)\in\mathbb{R}^n \mid -1\leq c_i\leq 1,\ i=1,\ldots,n\}$
which is closed, 0-symmetric, convex, and

has a volume $v(P) = 2^n$. P obviously contains lattice points $b \in \mathbb{Z}^n$ other than $(0,\ldots,0)$ and $\varphi^{-1}(b) \in o_\beta \setminus \{0\}$ holds. For $\omega \in o_\beta \setminus \mathbb{Z}$ there exists an integral matrix $\mathfrak{M} = (m_{ik}) \in GL(n,\mathbb{Z})$ with

$$(5)\quad \omega\beta^{i-1} = \sum_{k=1}^{n} m_{ik}\beta^{k-1} \quad (i=1,..,n)$$

and $\det \mathfrak{M} = N(\omega)$. Therefore we get $\varphi(\omega(b_1+b_2\beta+\ldots+b_n\beta^{n-1})) = (b_1,\ldots,b_n)\mathfrak{M}$ and the parallelotope

$$P_\omega := \{(c_1,\ldots,c_n)N(\omega)^{-1/n}\mathfrak{M} \in \mathbb{R}^n \mid -1 \le c_i \le 1,\ i=1,\ldots,n\}$$

is derived from P by a linear transformation of determinant 1. Then P_ω also contains a lattice point b_ω different from $(0,\ldots,0)$ by Minkowski's Theorem, and $\varphi^{-1}(b_\omega) \in o_\beta \setminus \{0\}$ holds. By considering parallelotopes P_ω for several ω so that any one of them is never a rational multiple of another we get as many elements of $o_\beta \setminus \{0\}$ as we like. It only remains to show that the norms of all these elements are bounded. But this follows immediately from the fact that the set

$$\{\prod_{i=1}^{n}(c_1+c_2\beta^{(i)}+\ldots+c_n(\beta^{(i)})^{n-1}) \mid -1 \le c_j \le 1,\ j=1,\ldots,n\}$$

is bounded in $\mathbb{R}$ and that transformations by numbers $\omega N(\omega)^{-1/n}$ do not affect the norm.

We order the elements of $P \cap \mathbb{Z}^n = \{(b_1,\ldots,b_n) \in \mathbb{Z}^n \mid b_i \in \{0,1,-1\},\ i=1,\ldots,n\}$ lexicographic and consider their images by φ^{-1} one after another. Let us assume that we have already examined g numbers and have stored l of them with norm not equal to ± 1 into an array A and have already found $m<n-1$ independent units $\eta_1,\ldots,\eta_m$. When we examine the next element z of $P \cap \mathbb{Z}^n$, then either $\varphi^{-1}(z)$ itself or $\varphi^{-1}(z)$ divided by some element of A might be a unit η. In this case we must check whether $\eta_1,\ldots,\eta_m,\eta$ are independent. By taking the logarithms of the absolute values on both sides of (3) in each field $F^{(j)}$ $(j=1,\ldots,n-1)$ we obtain a homogeneous linear system of equations with the matrix of coefficients

$$\mathfrak{A} := \begin{pmatrix} \log|\eta_1^{(1)}| & \ldots & \log|\eta_m^{(1)}| & \log|\eta^{(1)}| \\ \vdots & & \vdots & \vdots \\ \log|\eta_1^{(n-1)}| & \ldots & \log|\eta_m^{(n-1)}| & \log|\eta^{(n-1)}| \end{pmatrix}.$$

We determine the rank of $\mathfrak{A}^t$ (i.e. the rank of $\mathfrak{A}$) by Gauss's method. Since $\eta_1,\ldots,\eta_m$ are independent we assume that we have already stored the matrix

$$\mathfrak{A}_m := \begin{pmatrix} a_{11} & \ldots & a_{1m} & \ldots & a_{1n-1} \\ & \ddots & \vdots & & \vdots \\ 0 & & \ddots & & \vdots \\ & & a_{mm} & \ldots & a_{mn-1} \end{pmatrix}$$

$(a_{11}\cdot\ldots\cdot a_{mm} \neq 0)$, which originates from the first m rows of $\mathfrak{A}^t$ by the elimination method of Gauss. The rank of $\mathfrak{A}$ is equal to the rank of the matrix

$$\begin{pmatrix} & \mathfrak{A}_m & \\ \log|\eta^{(1)}| & \ldots & \log|\eta^{(n-1)}| \end{pmatrix}$$

and therefore $\eta_1,\ldots,\eta_m,\eta$ are independent, if $a_{m+1m+1} := \log|\eta^{(m+1)}| - \sum_{i=1}^{m} \frac{\log|\eta^{(i)}|}{a_{ii}} a_{im+1} \neq 0$ holds. In this case we compute the values a_{m+1j} $(j=m+2,\ldots,n-1)$ and store $\mathfrak{A}_{m+1}$. This method also enables us to compute the regulator R_η easily as the product $a_{11}\cdot\ldots\cdot a_{n-1n-1}$ as soon as we have found $n-1$ independent units $\eta_1,\ldots,\eta_{n-1}$.

When we have considered all elements of $P \cap \mathbb{Z}^n$ in this way and have not yet found $n-1$ independent units, we must transform P by some $\omega \in o_\beta \setminus \mathbb{Z}$ and carry on the procedure with $P_\omega \cap \mathbb{Z}^n$ instead. To determine $P_\omega \cap \mathbb{Z}^n$ we construct a unimodular matrix $\mathfrak{U}$ (together with $\mathfrak{U}^{-1}$) by the elementary divisor theorem which transforms the matrix $\mathfrak{M}$ of (5) into a lower triangular matrix $\mathfrak{N} := \mathfrak{M}\mathfrak{U}$. Also by elementary integer arithmetic we obtain a lower triangular matrix $\mathfrak{H}$ from $\mathfrak{N}\mathfrak{H} = N(\omega)I_n$ (I_n denotes the $n\times n$ unit matrix). We substitute $(f_1,\ldots,f_n) = (c_1,\ldots,c_n)N(\omega)^{-1/n}\mathfrak{M}\mathfrak{U}$ and get $(f_1,\ldots,f_n)\mathfrak{H} = N(\omega)^{(n-1)/n}\cdot(c_1,\ldots,c_n)$. Therefore every element of P_ω corresponds to a solution $(f_1,\ldots,f_n) \in \mathbb{R}^n$ of

(6) $-|N(\omega)|^{(n-1)/n} \le \sum_{j=1}^{n} b_{ji} f_j \le |N(\omega)|^{(n-1)/n}$

$(i=n,n-1,..,1)$,

where the b_{ji} are the entries of the matrix $\mathfrak{B}$. But these inequalities can be solved one after another each involving only one unknown. The set $P_\omega \cap \mathbb{Z}^n$ then consists of the vectors $(f_1,...,f_n)\mathfrak{U}^{-1}$, where $(f_1,...,f_n) \in \mathbb{Z}^n$ runs through all integer solutions of (6). Again $P_\omega \cap \mathbb{Z}^n$ is a finite set. We put it into lexicographic order and replace $P \cap \mathbb{Z}^n$ by it.

After having discussed the important features we present our method in algorithmic form and add some remarks afterwards. We assume that the set $P \cap \mathbb{Z}^n$ under consideration consists of r elements $z_1,...,z_r$ and that we have stored l numbers into A, the corresponding absolute values of the norms into an array NA, and have found independent units $\eta_1,...,\eta_m$, $m<n-1$ (stored into $E(1),...,E(m)$). Let $z_1,...,z_g$ already be examined. So we start with $l=m=g=0$. Then the algorithm works as follows:

Algorithm 1:

1. Replace g by g+1. If g>r, go to 14.
2. Set $x=\varphi^{-1}(z_g)$. If $x \in \{-1,0,1\}$, go to 1. Else compute $n_x=|N(x)|$.
3. If $n_x>1$, go to 7.
4. Check, if $\eta_1,...,\eta_m,x$ are independent. If they are not, go to 1.
5. Replace m by m+1 and store x into E(m).
6. If m=n-1, terminate. Else go to 1.
7. Set j=1.
8. If j>1, replace l by l+1, store x into A(l), store n_x into NA(l) and go to 1.
9. If n_x=NA(j), go to 11.
10. Replace l by l+1 and go to 8.
11. Set y=A(j)/x. If $y \notin o_\beta$, go to 10.
12. If $y \in \{-1,1\}$, go to 1.
13. Carry out steps 4,5,6 with y instead of x. The jump in steps 4,6 is to 10 instead of 1.
14. Replace $P \cap \mathbb{Z}^n$ by $P_\omega \cap \mathbb{Z}^n$, r by $\#P_\omega \cap \mathbb{Z}^n$, ω by ω^2, g by 0. (We again consider a set $\{z_1,...,z_r\}$, but with different numbers $r,z_1,...,z_r$.) Go back to 1.

As input data for our program we need the field degree n, the coefficients $a_1,...,a_n$ of the generating polynomial (1), the bounds $(1,...,1)$, $(-1,...,-1)$ of $P \cap \mathbb{Z}^n$, and a number $\omega \in o_\beta \setminus \mathbb{Z}$ to transform P in step 14. Then we can already compute the roots $\beta=\beta^{(1)},...,\beta^{(n)}$ of (1), the values $(\beta^{(i)})^j$ $(i=1,...,n;\ j=1,...,n-1)$, and the coefficients t_{ij} of $\beta^{n-1+i}=t_{i1}+t_{i2}\beta+...+t_{in}\beta^{n-1}$ which we use to form the integer product of two numbers of o_β. Of course we can also determine these values in advance and add them to the input data.

We conclude this paragraph with some remarks about possible refinements of the program. Since $P \cap \mathbb{Z}^n$ (and also $P_\omega \cap \mathbb{Z}^n$) is O-symmetric, we can eliminate all those vectors the highest non vanishing coefficients of which are negative. Before carrying out the computations in step 4. it is recommendable to examine, if x is equal to $\pm\eta_i$ $(i \in \{1,...,m\})$. If we must store a great number of elements into the arrays A, NA, it will be of advantage to order them according to the size of the absolute value of the norm. Then steps 8.-10. will become more efficient.

An important refinement of our program is a better condition for the termination. The obtained n-1 independent units may not be suitable for §4. In fact, we want independent units which will probably turn out to be fundamental units. A measure for the quality of our system is the size of the regulator R_η in connection with our lower bound (4) for R. If we suppose that the determined independent units are not all fundamental units, we will carry on the procedure. Every time a new unit η is found we examine whether or not it is already

contained in the group generated by $-1,\eta_1,\ldots,\eta_{n-1}$. If it is not, we determine a new system of generators $\eta_1',\ldots,\eta_{n-1}'$, for which $\langle -1,\eta_1,\ldots,\eta_{n-1},\eta\rangle = \langle -1,\eta_1',\ldots,\eta_{n-1}'\rangle$ holds. This can be done by the following algorithm.

Algorithm 2:

1. Determine $m_1,\ldots,m_n\in\mathbb{Z}$ from $m_1 \log|\eta_1^{(j)}|+\ldots+m_{n-1}\log|\eta_{n-1}^{(j)}|+m_n\log|\eta^{(j)}|=0$ and set $p_j=|m_j|$ $(j=1,\ldots,n)$.
2. Compute $p:=\min\{m\in\{p_1,\ldots,p_n\}\mid m\neq 0\}$ and let $j\in\{1,\ldots,n\}$ be maximal with $p=|m_j|$.
3. Set $i=1$.
4. If $i=j$, go to 6. Else divide m_i by m_j. We get integers $k_{ij}=[\frac{m_i}{m_j}]$ and l_{ij} so that $m_i=k_{ij}m_j+l_{ij}$ holds. If $l_{ij}=0$, go to 6.
5. Replace η_j by $\eta_j\eta_i^{k_{ij}}$, m_i by l_{ij}, p_i by $|l_{ij}|$ and go to 2.
6. Replace i by i+1. If $i\leq n$, go to 4.
7. Set $\eta_i'=\eta_i$ $(i=1,\ldots,n;\ i\neq j)$, $\eta_j'=\eta$ and terminate.

We must still explain in which way we get an integer solution in step 1. When we solve the given homogeneous system of linear equations, the solutions form a one dimensional vector space $V\subset\mathbb{R}^n$. We choose the vector $v=(m_1',\ldots,m_{n-1}',1)$ as a basis. Then we know from §2 that $m_1',\ldots,m_{n-1}'$ are rational numbers, the common denominator of which is bounded by $\frac{R_\eta}{R}$. The upper bound T for $\frac{R_\eta}{R}$ of §2 allows us to compute $m_1,\ldots,m_n$, as we determine the smallest integer k of $\{1,\ldots,[T]\}$ for which $kv\in\mathbb{Z}^n$ holds.

If the obtained system of independent units $\eta_1,\ldots,\eta_{n-1}$ does not change any longer, we expect that we have already found a system of fundamental units.

§4 Computation of fundamental units

The fundamental theorem for finitely generated abelian groups leads to a criterion to decide whether n-1 independent units $\varepsilon_1,\ldots,\varepsilon_{n-1}$ already form a system of fundamental units [2]:

Lemma: If $\varepsilon_1,\ldots,\varepsilon_k$ $(0\leq k\leq n-2)$ belong to a system of fundamental units of F, then the unit ε_{k+1} also belongs to this system, if and only if there is no $\eta\in o_\beta$ such that

(7) $\varepsilon_{k+1}=\pm\varepsilon_1^{m_1}\cdot\ldots\cdot\varepsilon_k^{m_k}\eta^m$ $(m\in\mathbb{N},\ m\geq 2;\ m_i\in\{0,-1,\ldots,-m+1\},\ i=1,\ldots,k)$.

We note that in case k=0 the lemma yields a criterion, if ε_1 actually belongs to a system of fundamental units. For k=0 equation (7) becomes $\varepsilon_1=\pm\eta^m$. Our estimate T of the group index $(E:G_\eta)$ of §2 enables us to restrict m to $2\leq m\leq T$. Moreover it suffices to look for solutions of (7) for prime numbers m only.

We give two simple methods to determine possible solutions of (7) which both work very well, if n and T are small enough. In the case n and T are large one should prefer a p-adic method [4] or at least some refinements recommended in [2]. We consider (7) in all $F^{(i)}$ and obtain a system of equations

(8) $\eta^{(i)m}=\pm\varepsilon_1^{(i)-m_1}\cdot\ldots\cdot\varepsilon_k^{(i)-m_k}\varepsilon_{k+1}^{(i)}=:\pm v^{(i)}$ $(i=1,\ldots,n)$.

A possible solution $\eta\in o_\beta$ is of the form

(9) $\eta=e_1+e_2\beta+\ldots+e_n\beta^{n-1}$ $(e_i\in\mathbb{Z},\ i=1,\ldots,n)$.

We note that comparing signs on both sides of (8) shows $m\neq 2$ in general. Therefore let m be odd, $m\geq 3$ further on. The case m=2 requires minor modifications of method 1.

Method 1:

For all possible values of m_i $(i=1,\ldots,k)$ (i.e. m^k cases) we compute

$e_j=\mathrm{Tr}(\bar\beta_j\eta)=\sum_{l=1}^{n}\bar\beta_j^{(l)}\sqrt[m]{v^{(i)}}$ $(j=1,\ldots,n)$

in double precision. If the e_j are not "integers", equation (8) is impossible. Otherwise we obtain a solution η and replace ε_{k+1} by η.

Method 2:

By taking the absolute values on both sides of (8) we get bounds S_i for $|\eta^{(i)}|$ by the estimates

$$\sqrt[m]{|\varepsilon_j^{(i)-m_j}|} \leq \begin{cases} 1 & \text{if } |\varepsilon_j^{(i)}|<1 \\ \sqrt[m]{|\varepsilon_j^{(i)}|^{m-1}} & \text{if } |\varepsilon_j^{(i)}|>1 \end{cases} \quad (j=1,\ldots,k).$$

By multiplying η with elements of the dual basis we obtain bounds T_j for the coefficients e_j:

$$|e_j| = |\mathrm{Tr}(\bar{\beta}_j\eta)| \leq \sum_{l=1}^{n} |\bar{\beta}_j^{(l)}| S_l =: T_j.$$

This leaves us with a finite number of possibilities for η. We compute $\eta^{(i)}$ $(i=1,\ldots,n)$ and eliminate η, if $|\eta^{(i)}|>S_i$ holds for some i or $N(\eta)\neq\pm1$. Otherwise we solve $m\log|\eta^{(i)}|+m_1\log|\varepsilon_1^{(i)}|+\ldots+m_k\log|\varepsilon_k^{(i)}|=\log|\varepsilon_{k+1}^{(i)}|$ $(i=1,\ldots,n)$ derived from (8). If the solutions $m,m_1,\ldots,m_k$ are integers, we replace ε_{k+1} by η.

Both methods use techniques which we already discussed in §2, §3. There are no difficulties in programming them. The methods must be carried out for $k=0,1,\ldots,n-2$ and all primes m, $3\leq m\leq T$. Obviously we prefer method 1, if k is small. For larger values of k we note that we can execute method 2 for several k in each step simultaneously, since the bounds T_j will grow with k. Of course this is recommendable only, if we are convinced that our units already form a system of fundamental units.

§5 Examples

a) Let n=5, β be a zero of $x^5+x^4-4x^3-3x^2+3x+1=0$. We have $o_\beta=o_F$ in $F_5=\mathbb{Q}(\beta)$. Algorithm 1 yields units

$\eta_1=3+\beta-7\beta^2+2\beta^4$, $\eta_2=-1-3\beta+\beta^3$,
$\eta_3=\beta^3+\beta^4$, $\eta_4=-1+\beta^3+\beta^4$

with regulator $R_\eta=16.357$. Since (4) provides $R>0.348$ we proceed including algorithm 2 and obtain after a few steps

$\eta_1'=-1-3\beta+\beta^2+\beta^3$, $\eta_2'=-23-99\beta-57\beta^2+30\beta^3+20\beta^4$,
$\eta_3'=\beta^3+\beta^4$, $\eta_4'=-1+\beta^3+\beta^4$

with regulator $R_{\eta'}=1.636$. Therefore the index of $\eta_1',\ldots,\eta_4'$ in a system of fundamental units of F_5 is less or equal 4. So we apply method 1 of §4 for m=3 and k=0,1,2,3 (40 cases). $\eta_1',\ldots,\eta_4'$ turn out to be fundamental units in F_5.

b) Let n=6, β be a zero of $x^6+x^5-7x^4-2x^3+7x^2+2x-1=0$. Again we have $o_\beta=o_F$ in $F_6=\mathbb{Q}(\beta)$. Algorithms 1 and 2 provide

$\eta_1=-1+3\beta-6\beta^3+\beta^4+\beta^5$, $\eta_2=\beta-\beta^2-6\beta^3+\beta^4+\beta^5$,
$\eta_3=3+7\beta-2\beta^2-7\beta^3+\beta^4+\beta^5$, $\eta_4=\beta$, $\eta_5=1+\beta-\beta^2+\beta^4$

with regulator $R_\eta=3.278$. Because of (4) $R>0.328$ holds, the index of $\eta_1,\ldots,\eta_5$ in a system of fundamental units is less or equal 9. We apply method 1 of §4 for m=3,5,7 and k=0,1,2,3,4 (3703 cases). $\eta_1,\ldots,\eta_5$ are a system of fundamental units of F_6.

We conclude with some remarks. The fields F_5, F_6 are totally real algebraic number fields of minimum discriminant of the degrees n=5,6 respectively. We suppose that their regulators are also small which shows that the lower bounds (4) for the regulators of fields are inadequate. I have tried to get better estimates but have succeeded only in some special cases so far. For other possibilities to estimate the group index $(E:G_\eta)$ see [4].

We recognize that the computations for low degrees n are very simple. They can be carried out in many cases by a pocket calculator for $n\leq4$. On the other hand the difficulties grow very fast with n, as the application of method 1 in a) and b) shows. Therefore I hardly see a chance to handle field degrees n>10 with these methods.

References

[1] Nagell, T.: Zur algebraischen Zahlentheorie, Math.Z. 34 (1931), 183-193.

[2] Pohst, M.: Invarianten des total reellen Körpers siebten Grades mit Minimaldiskriminante, to appear in Acta Arithmetica.

[3] Pohst, M.: Eine Regulatorabschätzung, to appear.

[4] Pohst, M. and Zassenhaus, H.: An effective number geometric method of computing the fundamental units of an algebraic number field, to appear in Math. of Comp.

[5] Stoer, J.: Einführung in die Numerische Mathematik I, Springer-Verlag 1972.

[6] Zassenhaus, H.: On Hensel Factorization II, Symposia Mathematica, Volume XV (1975), 499-513.

On the Use of Computer Programs for the Investigation of Kronecker Classes

by Rudolf Land

In his recent work W. Jehne introduced and investigated classes of Kronecker equivalent extensions of finite algebraic number fields (see [2] and [3]).

Following an idea of L. Kronecker [5] the Kronecker sets of polynomials are introduced: for rational irreducible polynomials f with integer coefficients the set A(f) of primes modulo which f has a congruence root,
$A(f):=\{p \text{ prime} \mid f(x)\equiv 0 \bmod p \text{ solvable}\}$
is called the Kronecker set of the polynomial f. With this notation we call two polynomials f and g Kronecker equivalent $f\sim g$, iff $A(f)='A(g)$, where "='" means equality up to finite sets. In particular two polynomials are Kronecker equivalent, if they generate isomorphic fields (by adjoining one root respectively). Hence the notion of Kronecker equivalence for polynomials leads to a corresponding notion of number fields.

The aim is the study of the classes of Kronecker equivalent number fields, the so called Kronecker classes. Among Galois extensions the Kronecker equivalence relation coincides with the equality relation (theorem of Bauer, [6], [8]).

One can easily translate the Kronecker equivalence of polynomials into a condition on their Galois groups. This leads to an equivalence relation for subgroups of a given abstract (finite) group G, the G-equivalence: per definition two subgroups U, U' are G-equivalent, iff ${}_GU={}_GU'$, where ${}_GU:=\bigcup_{\sigma\in G} U^\sigma$ denotes the G-cocore of U. Hence G-equivalence amounts to an individual conjugacy of the elements of U to the elements of U'. The main problem is the determination of all G-equivalence classes of subgroups of G considered as a graph with respect to inclusion. This reduces to the task of computing the cocores ${}_GU$ of subgroups U of G. Besides results of general nature in [2], [3], [4] we are interested in numerical calculations for groups of low order.

There are several computer programs initiated by J. Neubüser and J. Cannon for the investigation of groups (a description see for instance in [1] and [7]). Among these one program, which is quite well adapted for our problem, even provides a full determination of the subgroup lattice and the conjugacy classes of subgroups. In this connection we are indebted to J. Neubüser for kindly placing this program to our disposal. Since december 1975 it is implemented on our machine in Cologne, a CD CYBER 76. Handling our problem we are going to change and to enlarge that program in order to calculate the cocores ${}_GU$ for all nonconjugate subgroups U of G. After the systematical investigation of the groups of order less than 100 special groups will be treated, coming from special extensions of algebraic number fields.

REFERENCES

[1] J. CANNON: Computers in Group Theory: a Survey. Comm. ACM, vol. 12, number 1, Jan. 1969, 3-12.

[2] W. JEHNE: Kronecker classes of algebraic number fields. To appear in J. of Number Theory.

[3] W. JEHNE: On Kronecker classes of atomic extensions. To appear in Proc. Lond. Math. Soc.

4 N. KLINGEN: Atomare Kroneckerklassen mit speziellen Galoisgruppen. To appear in Abh. Math. Sem. Hamburg, 44.

[5] L. KRONECKER: Über die Irreductibilität von Gleichungen. Werke II, 85-93.

[6] LEWIS-SCHINZEL-ZASSENHAUS: An extension of the theorem of Bauer and polynomials of certain special types. Acta Arith. 11 (1966), 345-352.

[7] J. NEUBÜSER: Investigation of finite groups on computers. Proc. Conf. on Comput. Algebra, Oxford, August 1967, Pergamon New York, 1968.

[8] A. SCHINZEL: On a theorem of Bauer and some of its applications. Acta Arith. 11 (1966), 333-343, 22 (1973), 221-231.

Design Features of COBALG

J P Fitch, P Herbert, A C Norman
University of Cambridge Computer Laboratory

Summary

The COBALG project grew from two somewhat conflicting aims. The first was a desire to investigate techniques for handling algebraic expressions too large to fit conveniently in primary memory; the second was to provide a striking, but reasonably light hearted project for a post graduate student. The first of these aims dictated that the algebraic expressions will be treated as basically disc rather than store resident. With this in view the second aim was satisfied by the picturesque choice of COBOL as an implementation language.

As far as we know we now have the first ever COBOL based algebra system, and furthermore the package owes more of its pholosophy to the SCHOONSCHIP system than to more standard shaped systems. The object of this paper is to discuss some of the consequences of building a system in this admittedly curious fashion, and to demonstrate that our strategy can lead to extremely practical results.

Introduction

Experience with a number of different computers and a variety of algebra systems suggests that, at least in Europe, the free use of symbolic calculations is still being held back by limitations of primary memory. Typically on a PDP10 daytime users are limited to about 35K words; on System 360s jobs needing more than 250K bytes are considered large. In Cambridge we have a 3M byte 370/165 which the maximum partition ever allowed is 400K bytes. These storage limits are at an embarassing level: most of the readily available algebra systems can run in 250K bytes, but will only be able to tackle fairly small calculations in such a region. Storage problems become even more extreme when it is desired to use a mini computer, when it becomes only too easy to fill the whole of a 64K byte machine with code, leaving no room for data at all.

Miola and Cioni [5] have shown what can be done with a small machine, a shoehorn, paging and a machine code programmer, by implementing a useful amount of SAC-1 on a mini computer. We wish to sidestep the difficulties they faced, while still fitting a full quart algebra system into a bare pint machine.

Initial Decisions

Since our main interests are in solving problems that could not realistically be worked by hand we were not prepared to consider a system with a tight limit on the size of expressions that could be handled. Again, despite recent advances large virtual stores are not readily available to all, and in particular to us. We therefore decided to follow the lead of SCHOONSHIP [7] and work on the basis that all expressions are kept as sequential files on secondary store. There are two important features of the simple class of algebra we wanted to support that make this approach feasible:

1) all the important algorithms needed access their data inputs in a basically sequential manner,

and 2) the major non sequential process needed is sorting; and disc based sorting algorithms have been studied and developed extensively.

To keep the first of these features true our system is restricted to working with Poisson series. We do not feel that, for instance, rational function manipulation would fit cleanly into our disc based framework. To take full advantage of the technology developed for the second of the features, and to enable us to get an experimental system going rapidly we decided to implement our new system in ANSI standard COBOL [1] which has a SORT statement. Hence the name COBALG. None of us had written a non trivial COBOL program before, and so initially our choice of language was based on a mixture of curiosity and the book knowledge that the COBOL language was designed to make the handling of files fairly straightforward. In the belief that many academic or scientific programmers only know of COBOL by hearsay, we feel that it is worthwhile listing some of the features of the language that we have found to be of particular help. These features are all part of the ANSI Standard, and so can be expected to be available fairly generally.

1) There are facilities in the language for defining, opening, closing, rewinding and generally manipulating arbitary data and work files. The programmer can select suitable record lengths and blocking factors for each file. While equivalent capabilities may exist in other languages, in COBOL

the file control mechanism is both general and standard.

2) Standard COBOL provides a SORT statement that sorts a file into an order defined by arbitrary fields within records. In many COBOL systems this statement provides a direct link to a highly optimized sorting package which will probably have higher performance than any code we would be likely to write. For example on IBM360 the sorting is done by the excellent SORT/MERGE package [4]. Furthermore the SORT statement allows user written procedures to preprocess and postprocess the data. In our case this is particularly useful as the post process can easily detect and merge terms in an expression which have the same exponent sets.

3) Numeric fields can be up to 18 (decimal) digits wide, and so for many purposes there is no need to provide any kind of multiple precision arithmetic package.

4) Arbitrary width fields of data within records can be compared or moved with ease. This means that in many places a single COBOL statement serves the purpose of what we would normally have to write as a loop. A particular example of this in action is the comparision of exponential sets in the post processor mentioned above.

5) Overlays are catered for in a straightforward way, by building the structure into the source text. This avoids complex use of linkage editors and the like, and the programmer is encouraged to think in terms of segmented code from the very start, and so perhaps stands a chance of writing a program that will fit into small machine.

To balance these positive points it is fair to note that the COBOL language is rather verbose, being based largely on normal English; it is unduely sensitive to the presence or absence of full stops, and is lacking many of the control structures to which we have become used in the languages of Computer Science.

It is also appropriate to mention the other languages that we might have used, but discarded. Almost all possibilities are ruled out from the start by the requirement that transput facilities should be comprehensive and efficient: on our IBM370/165 the only plausible alternatives to COBOL left by this constraint are machine code and PL/I. We avoided the first of these because we wanted a flexible experimental system, and could not afford the investment in a substantial machine code, and non transferable, program. The reasons for choosing COBOL rather than PL/I were rather more marginal. The decision was made in view of a feeling that a COBOL system would be more portable, more suitable for small machines and that using COBOL would force us to concentrate our energies on developing a true disc based system. PL/I is such a comprehensive language that there would be a danger that the initial limited project would get lost in compromises about the exact data structures, hybrid main store and disc representations and other unwarranted generality.

The results to be presented later should give some idea of the advantages gained and the penalties paid in making these gross design decisions.

Implementation and Results

The major decision that had to be made in filling out the gross design framework presented above was that of defining precisely what class of expressions were to be handled. This has to be worked in close conjunction with the specification of major data structure formats and the use of records on disc.

The system finally produced is a Poisson series processor, and so all expressions are built up from terms of the form

$$\pm(P/Q)\ a^{l} b^{m} c^{n} \ldots h^{s} \cos(iu+jv+kw)$$

where j,j,k,...s are all integers. a,b,...h are polynomial variables, while u,v and w are harmonic variables. As well as cosine terms (as illustrated above) there are of course sine terms. Note particularly that the variables in the polynomial and harmonic parts of the expression are quite separate, so that x sin(x) for example cannot be handled. Furthermore the arguments for sines and cosines are never more complex than sums of harmonic variables with small integer coefficients. When manipulating Poisson series the only simplification rules needed are those for collecting terms and those that linearize products of harmonic terms like

$$\sin(u)\ \sin(v) \rightarrow (1/2)\ (\sin(u+v) + \sin(u-v)).$$

In COBALG the term is represented as a packed record. The coefficients P and Q are represented as 18 digit fields, the exponents l,m,...s are two digits each, while the multipliers i,j and k associated with harmonic variables are given space for two digits plus a sign. The declaration of a suitable record is given in figure 1. Since an expression is represented by a sequential file of terms, no chaining words or freestore management routines are needed.

In terms of this data format it is easy to implement addition and subtraction, integration and differentiation, multiplication and division by a single term, as well as editing and and selection. Note that none of these operations call for recursion or any other programming technique or style that does not fit conveniently into the COBOL philosophy.

The only other operation needed for straightforward application is substitution, and non trivial cases of it involve a relatively complex sequence of operations. Horner's rule could be implemented by re-sorting the input expression using just one variable as the key, arranging the sort to leave the expression represented in descending powers of that variable. The expression could then be dismantled and recombined using nested multiplication. Our view is that this somewhat complex process is not particularly suited to doing simple substitutions on large expressions, and so instead we do the

```
01  POLYNOMIAL-TERM.
    02 SIGN PIC X.
    02 COEFFICIENT.
       03 NUMERATOR PIC 9(18).
       03 DENOMINATOR PIC 9(18).
    02 SORTED-ITEMS.
       03 EXPONENTS.
       04 EXPONENT1 PIC 99 OCCURS 6 TIMES.
       04 EXPONENT2 PIC 99 OCCURS 6 TIMES.
    03 SIN-COSINE-PART.
       04 SIN-COSINE PIC A.
       04 ANGLES.
          05 HARMONIC-COEFFICIENTS PIC S99
                OCCURS 6 TIMES.
```

Figure 1: A COBOL record structure for a polynomial term.

substitution term by term, leaving the general sorting algorithm to tidy up afterwards.

One particular application that we had in mind when considering building COBALG was the solution of large sets of linear equations. Experiments using CAMAL [2] had suggested that for some problems at least Smit's method [6] could be extremely effective. This method is almost ideal for a disc based system. A fairly small driver program generates successive terms in a determinent, and eventually has to get these sorted and merged. Although there are some doubts about the utility of inverting large symbolic matrices, attempting to do so seems to have become a popular activity [3,6]. Using CAMAL we were strongly store limited, but found that even the largest problems with which we could cope only took a few seconds of machine time. COBALG clearly lets us go much further; it may be somewhat slower than normal systems for problems we have in common, but its ability to handle almost arbitrarily large expressions can more than compensate for this.

Of course to make COBALG into a working system we have a compiler for it, written in COBOL, that translates a simple autocode like language into interpretive code for the runtime system.

Conclusions

COBALG points the way for a new family of algebra systems for smaller machines. It faces directly up to the fact that expressions can get very large very rapidly, and that main storage limitations are normally rather inflexible, whereas cpu cycles and disc access are now quite cheap. COBOL is not altogether ideal for this project, as it is not possible to index files, and so much duplication of code is found. Despite this we have shown that there may be genuine scientific uses for COBOL, and that as the business community has clearly known for a long time, the language does have its good points.

References

1 ANSI COBOL Standard X3.4

2 J P Fitch "CAMAL User's Manual" University of Cambridge Computing Service, 1975

3 M Griss University of Utah - Computational Physics Reports UCP-28 and UCP-32

4 OS SORT/MERGE Programmer's Guide, IBM SC33-4007-5

and OS SORT/MERG/ General Information. IBM GC33-4022-1 (Program Number 57)

5 G Cioni and A Miola, Joint NIGSAM/SEAS SMC Project meeting London, 1976

6 J Smit, these proceedings

7 H Strubbe, Manual for SCHOONSCHIP, Computational Physics Communications 8(1974)

Basic Algorithms in FORMAC: Design and Verification

Knut Bahr
GMD/IFV
D-6100 Darmstadt, Germany

Abstract

The paper discusses a few basic algorithms used in FORMAC for handling common subexpressions, adding and multiplying sums, and raising sums to a power. It presents some data on their verification and improvements achieved in the latest version of FORMAC73.

1. Introduction

FORMAC is a well-known system for dealing with algebraic expressions, originally developed by a group within IBM as a pioneer effort in the sixties [21]. In the last years, it has undergone a major revision and a series of tunings, resulting in a considerably more efficient version, called FORMAC73 [see e.g. 1,2,3]. This is a report on attempts to introduce some of the more recent fundamental algorithms into FORMAC73.

The approach taken is to study theoretical and empirical results on various algorithms reported in the literature, to select one or more for implementation in FORMAC, and to measure and verify their performance in this environment. Not always is it practical or feasible in the given framework to implement the theoretically most prominent algorithm (especially if it is best only for a specific class of problems). So, we modestly aimed at algorithms that appeared to be among the better ones for a wide range of applications.

A great number of tests and measurements has been carried out. Rather than giving an almost complete set of test runs, we have selected only a few problems, which are meant to exemplify a typical behavior. Once again, activity counts play a major role in measuring and tuning [2]. In CAMAL, the same technique has led to improvements much like the ones we have experienced in FORMAC in the past [10]. So, having sensible measurement tools built in to a system seems at least as important as having good algorithms. Definitely, more studies are needed to develop the basic technique.

Many of the algorithms discussed have been developed for polynomials. FORMAC is not a polynomial system; it does not have a polynomial data structure like e.g. SAC-1[4], or "generalized" polynomial structures like REDUCE[13], or a polynomial subsystem like CAMAL[8,9]. Probably, it would have been a better choice to have a polynomial subsystem in order to exploit polynomial algorithms. As it stands, algorithms in FORMAC have permanently to be on the lookout for operators and non-polynomial components. Without a re-design of FORMAC, it is very difficult to estimate what the effect of a different data structure would be.

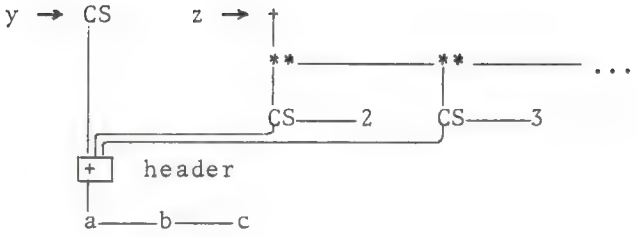
So, in the following we shall mainly refer to sums instead of polynomials, having in mind that terms like sin, exponential, arbitrary functions etc. may occur within them or that they may represent unexpanded, factored, or widely "unstructured" polynomials. In view of many FORMAC applications, we shall mostly put emphasis on sparse polynomials when we talk about polynomials.

2. Common Subexpressions

FORMAC uses a list structure with explicit operators for internal representation of symbolic expressions [19,21,22]. Whenever a copy of an already existing subexpression is needed in two or more places within a structure (or across several ones) it creates a common subexpression (CS): The subexpression is put under a header containing a reference counter; any references to it will be through a so-called CS-node pointing to the header.
For example

$$y = a + b + c \qquad \text{(E1)}$$
$$z = y^2 + y^3 + \ldots$$

has the following list structure:



Common subexpressions are logically transparent to FORMAC internal algorithms and invisible to the user.

Three improvements have been made with respect to utilization of CSs:

1) Create CSs whenever two identical structures are met and preserve them as long as possible;

2) Make sure repeated simpification of CSs is avoided whenever possible;

3) Utilize CSs to minimize evaluation of functions.

FORMAC does not trace out identical structures that may originate from different sources or computations; it would be wasted effort. However, whenever two subexpressions are compared (which occurs frequently enough during simplification and sorting) and are found to be equal (or parts of it are found to be equal) then a CS is generated. This was first suggested by J. Smit [20].

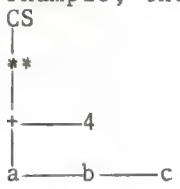
To save space, CSs should be preserved as long as possible and be replaced only when pertinent transformations are applied to them. For instance, in example E1 y should be kept as a CS, while in the sequence

$$y = a + c + d$$
$$z = y + b$$

a plus-transformation is applied to y and therefore within z, y is not kept as a CS (otherwise a sum structure might become rather unwieldy).

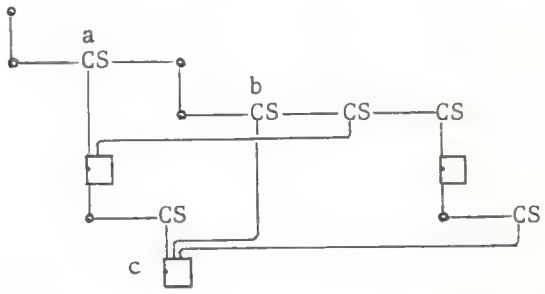
The automatic simplification routine AUT used to remove CSs too soon; it could not tell ahead of time if a prospective transformation would actually apply. Now, it has been greatly rewritten to acquire more contextual information and to include a look-ahead under CS-nodes. Thus, it no longer removes the CS-nodes in example E1, since it recognizes that none of the candidate transformations that exist for similar structures apply in this particular situation.

Improvements (2) and (3) concern time. Underneath a CS-node should always be a simplified, canonical structure so that generally on simplification of an expression CSs need not be reprocesses. However, since simplification may take place in different modes, each node, including any CS-node, carries bits telling in what mode the structure below has been simplified.
For example, the CS $(a+b+c)^4$



would carry the mode bit for NOEXPAND. If the mode is changed to EXPAND, in order to obtain $a^4 + 4a^3b + \ldots$, the CS should be expanded only once and the result substituted for each occurrence of the CS within the current structure. Of course, the algorithm has to be careful not to make any changes below a CS-node which might affect other list structures.

Given a more general structure, like



simplification is applied recursively to each CS that requires it and to the entire structure. The question arises: when we get to node b, how do we know that subexpression c has already been processed underneath node a.

The algorithm implemented at first, roughly worked as follows.

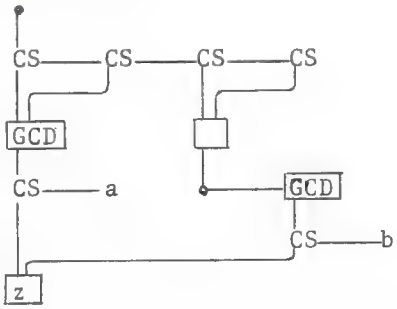
Algorithm A1

1. Use headers as temporary memory; assume all headers are initially marked "not already processed"; assume a header has an additional pointer to a new (simplified) value if it is marked "already processed".
2. Go down and across the structure.
3. If a CS is met in going down/across:
3.1 if CS simplified go on across;
3.2 else if CS not already processed go on down;
3.3 else pick up new value from header, substitute it for CS-node, go on across.
4. If a CS is met in coming up:
4.1 simplify the subexpression,
4.2 mark header already processed,
4.3 save new value in header,
4.4 break out all CS-nodes above,
4.5 go on across.

The actual algorithm also sets bits in each node to tell AUT what parts need simplification. There was a rather interesting flaw in it: Bits were also reset inside a CS before it was simplified in the current mode. In certain situations, this resulted in repeated unnecessary resimplifications of an already simplified CS. The interesting question is how does one discover such a defect. Notice that the flaw produces a side-effect on data structures, which may show up in arbitrary places. Thus, the common technique of splitting a problem in subproblems and analyzing the subproblems will not help in general. Also, monitoring of activities is of limited use, as typically the increased activity is well distributed and only by accident (in a special problem) will a suspicious peak show up. The only technique that helps is built-in redundancy, i.e. verification of preconditions. Corresponding code has been inserted and led to the discovery of the flaw. In this case, it is easy; whenever a presumedly simplified CS is encountered the system will test a bit to verify the assumption. In other cases, more overhead is involved. For instance in sorting, verifying the presumption that a term was already in the correct position by still sorting it in greatly helped to reveal oversights in the design. But this is a redundancy one would not want to leave in the system permanently.

Let us come back to algorithm A1. It has the following deficiency. Headers are used to recall "new" values; they hold a pointer pointing to the first occurrence of the new value in the current structure. If the new value in turn appeared under a CS-node, it may go away on simplification of the higher level CS and the header pointer would become invalid. For this reason, higher level CS-nodes are removed in line 4.4, thereby halting simplification of nested CSs. On the other hand, headers cannot hold a pointer to a copy of the new value, if there is no way of telling when it is no longer needed and should be returned to free storage (no garbage collection). Thus, we shift the memory from the headers into a separate list (see below), which holds pointers to CS-copies and can be accessed and released at the end. This however has a deficiency of its own (aside from overhead) in that it forces all new values into CSs. It seems, these imperfections can only be avoided by a pre-scan of the structure to gather information on the CSs; the gain is questionable.

What is more important is that the general scheme can be extended and adopted to delayed evaluation of functions [3]. FORMAC73 allows for the construction of indeterminate expressions like may be GCD (z, $y), read: the gcd of expression z and some other expression y. Evaluation of the presumedly user-supplied function GCD will be delayed until an actual y is given, e.g. GCD(z,a) or GCD(z,b). The system may, for instance, end up in a structure like



Clearly, GCD should be called no more than twice, here. Note that there are now unsimplified CSs: GCD(z,a) is an unevaluated expression and as such unsimplified by definition. No pointers from outside the current structure can point to them. Therefore, the new, simplified values may simply be substituted for the old ones in the headers and need not be memorized. However, we do need to recognize which CSs have not yet been processed. Remembrance bits cannot be placed in headers, as general recursive functions would reuse them. Thus, a separate memory is established in form of an ordered linked list, to keep track of already processed CSs.

To verify the algorithm, one merely needs to monitor the function calls. There is no point in giving examples. The mentioned memory had a nice side-effect: Augmented by a few details, it served as an excellent log in debugging the algorithm.

The described technique has proved very effective in reducing the computation time, especially when costly functions are involved. It may even be useful to extend the memory to keep track of all function calls, or to seek out common subexpressions. Further studies are needed to find out if situations justifying such a possibly expensive technique occur frequently enough in practical problems.

3. Multiplication of Sums

Various algorithms for multiplying polynomials have been given in the literature [e.g. 6,17,18]. FORMAC, in multiplying two sums with m and n terms, used to take the naive approach [17], i.e. generate all mn terms, simplify and sort them, merging like terms. In the following, we want to concentrate on the "divide and conquer" method, which actually aims at improving the addition of (intermediate) sums. It also helps in cutting down the space requirements. The remaining problem of multiplying a sum by a single term is discussed in section 4.

Following Johnson [17], we find that the sorting cost for the naive approach (call it algorithm C1) is of order

$$C1 = mn\log(mn)$$

For the divide and conquer approach we find: If C2(n) is the cost of sorting n sums of size m and if one employs a linear merge to add two sums, then we have the recurrence relation

$$C2(n) = 2\,C2(n/2) + mn$$

A general solution is C2(n) = mnlog(n) + an, where a is some factor that can be determined from some initial condition for n=1.

There is a tacit assumption in Johnson's paper:

(*) No re-sorting is required when a sum of size m is multiplied by a single term.

Therefore, we have a=0 and obtain

$$C2(n) = mn\log(n)$$

which is a improvement over C1.

The assumption (*) is natural for polynomial based systems (e.g. ALTRAN [12], SAC-1[4]), but not necessarily so for general systems like FORMAC (see section 3). Without it, we have to include sorts of order mlog(m) or set a=mlog(m), obtaining

$$C2(n) = mn\log(mn)$$

which is of the same order as C1.

This shows
1) the superiority of the divide and conquer method stems from its utilization of given ordering information;
2) even if this information is not utilized C2 is at least as good as C1.

To verify (2), the method was first implemented without the above assumption (*). It allows us to tackle the space problem in a convenient way.

Algorithm C2 (divide and conquer):

Given $\sum_{i=1}^{n} x_i \cdot \sum_{j=1}^{m} y_j$ for some x and y

define $z = \sum_{j=1}^{m} y_j$

as as a common subexpression with all y_j being common subexpressions.

Initialize r = 0 (resulting sum).

For i = 1 to n do:
generate $x_i \cdot z$; (no extra terms generated, z is not copied) recursively distribute x_i over z; (y_j not copied) merge the resulting terms into the final sum r, as they are generated;
end.

For dense polynomial multiplication, the algorithm is asymptotically mn with the assumption (*) and asymptotically mnlog(m) without assumption (*) , i.e. still better than C1. On the other hand, it still generates all possible terms, most of which combine, and it has considerable overhead over C1, in particular in sorting. Therefore, it is of some interest to measure its behavior. We set up the problem

$$z = 1 + x + x^2 + \ldots + x^{n-1} \qquad \text{(E2)}$$
$$w = x + x^2 + \ldots . x^{n-1}$$
$$r = z \cdot w$$

The results on time and space are given in Table T1.

We find that the times grow with order $n^2\log(n)$ for both C1 and C2; over all, times for C2 are about 20% less. The core space for C1 grows with order n^2, for C2 it grows linearly with n. So, that solves the space problem and helps a bit on time, too.

We turn to a simple sparse polynomial example. (E3)

Multiply for various n,
$(a_1 + a_2 + \ldots + a_n) \cdot (b_1 + b_2 + \ldots + b_n)$
where the a_i and b_i are distinct variables with a random relative order among any two a_i and b_j.

Simple problems as E2 and E3 enable one to theoretically derive exact figures or bounds that have to be met. In a further step of verification one would of course move on to more realistic and complex problems. As an example, we present measurement data for the well-known Sigsam problem 2, see [2] or SIGSAM Bulletin no. 22, April 1972.

Table T2 gives the number of comparisons in sorting sums, as measured for example E3, n=10, for different algorithms. Table T3 gives the corresponding figures for the Sigsam problem 2.

First, we compare E31 through E33.
E31 is for algorithm C1 without assumption (*);
E32 is for algorithm C2 without assumption (*);
E33 is for algorithm C2 with assumption (*).

We notice, the divide and conquer method does improve the sort even without assumption (*). The reason is, the given formula is really a worst case analysis, while the actual numer of comparisons depends on the relative order in which terms are presented to the sort. With assumption (*), we achieve the expected dramatic improvement.

For Sigsam problem 2, we find that algorithm C2 alone does not improve things at all (S22 vs. S21). Instead of running the problem with assumption (*), we took a closer look at the merge function in algorithm C2.

n	time C1	time C2	$n^2\log(n)$	core C1	core C2
10	180	140	332	136	134
15	440	350	879	140	136
20	940	780	1729	146	138
25	1570	1250	2902	154	140
30	2300	1840	4416	162	142
35	3230	2600	6283	174	144
40		3400	8515		148
45		4700	11121		150
50		6150	14109		152

Table T1

	E31	E32	E33	E34	E35	E36
compares E3, n=10	506	470	280	445	437	247

Table T2

compares Sigsam 2	S21	S22	S23	S24	S25	S26
n = 4	172	172		179	174	162
n = 5	542	546		568	516	450
n = 6	1601	1612		1691	1446	1168
n = 7	4427	4500		4757	3801	2832

Table T3

At this point, we have to reveal that so far we did not employ a linear merge for adding sums, rather a logarithmic merge. It is a straightforward generalization of the classical logarithmic insertion. There were no reasons for choosing it but programming reasons. While the linear merge needs a minimum of m and a maximum of n+m-1 comparisons, the minimum and maximum for the logarithmic merge are of order log(n) and mlog(n), respectively.

E34 in Table T2 gives the compares for the linear merge, without assumption (*). In this example, we find a slight improvement over E32. In other cases, we find just the opposite, for instance in the Sigsam problem 2 (S24 in Table T3).

There have been attempts to combine the merits of either algorithm into one. The most significant result, to our knowledge, is the algorithm by Hwang and Lin [16]. Following Knuth[18], we call it a binary merge. We give a modified version of it. Details of the implementation have been omitted, e.g. the establishment of the condition m≤n, optimization with respect to special cases, and efficient placing of tests.

Algorithm C3 (binary merge):

Calculate $\sum_{i=1}^{n} A_i + \sum_{j=1}^{m} B_j$, $m \le n$,
by inserting the B's among the A's,
given $A_1 \le \dots \le A_n$, $B_1 \le \dots \le B_m$.

```
    j:=1;  i:=0;
L1: if m=0 then stop;
    if n≤0 then
     {append remaining B's to end of A's;
      stop};
    if n≤m then q:=1;
           else q:=floor(n/m);
    if A[i+q]<B[j] then {n:=n-q; i:=i+q};
    else if A[i+q]>B[j] then
     {insert B[j] between [A[i],...,A[i+q]];
      k:=pos-of-B[j]-succr-after-insrt - 1;
      (i≤k<i+q, imagine backwd renumbring)
      n:=n-(k-i);  i:=k;
      m:=m-1;      j:=j+1 };
    else if A[i+q]=B[j] then
     {combine B[j] with A[i+q];
      n:=n-q;      i:=i+q;
      m:=m-1;      j:=j+1 };
    go to L1; ■
```

In C3, "insert" is the classical logarithmic insertion, "combine" may be considered a special case of it.

Using the binary merge with the divide and conquer method, gives a little improvement in example E3 (E35 without assumption (*), E36 with it) and a noticeable improvement for the Sigsam problem 2 (S25 without assumption (*), S26 with it).

Measurement actually comprises the monitoring of many more counts than just the one discussed, as one wishes to minimize other activities as well or at least make sure they do not spoil an effect by adverse behavior. For instance in the Sigsam problem 2, n=7, the described steps led also to a significant reduction of the following important activities:

compares on product sorting	from 2189 to 1711
numeric coefficient operations	from 3372 to 1902
subexpression copies	from 1680 to 1017

Measurement of counts allows for fine tuning. For instance, the difference between a centered and an uncentered logarithmic insert within C3 can be sensed as a roughly 2% difference in the compare counts for Sigsam problem 2.

The number of compares depends on the relative ordering of the variables in the sums. So, for example E3, we have tried different orders and measured for example
495, 437, 427, 414, 387, 378 .
The lowest value (378) is obtained when all variables are in order across the two chains ($a_n \cdots b_1$). When in this order a_1 and b_1 are exchanged, the highest value (495) is measured. It is easy to figure out that the theoretical minimum is 378. which has indeed been achieved. The theoretical maximum, according to Knuth is 521, which cannot be achieved by any arrangement of variables in this problem.

Applying the algorithm to dense polynomial multiplication, as in $(x+x^2+\ldots x^{10})\cdot(x^k+x^{k+2}+\ldots x^{k+10})$ for k=1,..,11 always results in the same compare count of 290 without assumption (*), or 100 with it.

Whereas Hwang and Lin's algorithm is symmetric, ours is not. That is, as the algorithm inserts B's among the A's, the tail chain of A's may eventually become shorter than the remaining B-chain. At this point, the chains should be switched, which is not done. The initial motivation for not doing so was the entailed complication of the program. Because FORMAC works with linked lists, considerable overhead is introduced in delinking and relinking sublists. This could be verified with a trial implementation of the symmetric binary merge. It is possible that the overhead would be subordinate in a system with different data structure. More important is that the number of compares did not go down significantly; on the contrary, it increased slightly for most problems tried. We do not have a good explanation yet; the phenomenon has not been studied in further detail. Nevertheless, we suspect that in certain cases the symmetric binary merge may cause an oscillation, where subchains are switched back and forth frequently, thereby at least increasing the overhead noticeably.

4. Multiplication of a Sum by a Term

In a straight polynomial system, distributing a monomial over a polynomial does not present any problems. In particular, it is not necessary to re-sort the resulting polynomial when the given polynomial is already sorted. In a general system, like FORMAC, distributing a term over a sum may generate terms of a new structure that do require re-ordering.

Definition: We say, a structure has "collapsed" if, in general, the new structure takes a different position in the ordering than the old one.

Examples:

1) $e^{(5/8)\pi i}\cdot(e^{(11/8)\pi i} + \ldots) = 1 + \ldots$
2) $x^2\cdot(x^3 + x^2 + \ldots) = x^5 + x^4 + \ldots = \ldots$
   if there is some built-in reduction rule, that, for some x, reduces $x^5 \to x$ or $x^5 \to y$; take for example $i^2 \to 1$.

We are concerned only with so-called Autsim-rules, i.e. transformation rules which are built in to the system and as such are automatically applied (regardless whether they can be turned on or off).

Let t be a term, s a sum of n terms s_i (i=1,...,n), r a sum of m terms r_i (i=1, ...,m).

The algorithm first used in FORMAC is:

Algorithm A2

```
dist(t,s): r = 0;
           for all s_i ∈ s do:
           r = merge (r, simplify (t·s_i))
           end.
```

Thus, each term s_i is multiplied by t, simplified, and then sorted into the result sum r, as it may require re-ordering. To improve sorting, the merge would first test if a new term r_i goes at the end of the current sum r. Logically, this is equivalent to

```
dist(t,s):
step1: for all s_i ∈ s do:
       r = replace in s, s_i with (t*s_i)
       end;
step2: for all r_i ∈ r do:
       r = re-sort(r, simplify(r_i))
       end.
```

This suggests, that step1 is all we need in the majority of cases (assuming that $(t*s_i)$ already includes trivial simplifications such as $x \cdot x = x^2$); step2 is merely a check on the correct ordering, but it is unhappily married with the general simplify. So, our goal was

1) to give the system the capability of deciding whether or not step2 is required;
2) to separate simplification from resorting and to derive, from the simplification process, information on the ordering.

It turned out that the realization of point (2) was a very fruitful breakthrough. The new algorithm is:

Algorithm A3

```
dist(t,s):
step1:
for all s_i ∈ s do:
r = replace in s, s_i with (t*s_i)_tag
end;

step2:
for all (r_i)_tag ∈ r and tag = NS do:
r = replace in r, r_i with simplify(r_i);
if collapsed(r_i) then re-sort(r,r_i)
end. ■
```

Step1 attaches a tag to each term r_i. Only when the tag reads "NS" (for "not sufficiently simplified") will terms be given to simplify and only when simplify determines that their structure has collapsed, will they be re-sorted.

The algorithm can be nicely checked out and verified step by step. First, always set tag=NS and collapse=TRUE; verify that it behaves exactly as algorithm A2; make sure the added overhead is negligible. Then, start setting true tags and gradually reduce the truth of "collapsed"; observe results, invocation counts, run times. This process is quite sensitive to truth and falsity of "collapsed". It also helped to reveal flaws in the theoretical outline.

As an example, we consider a small problem that is a slight modification of an idea by Fitch and Norman[7]:

Calculate

$$F_n(z) = z\,F_{n-1}(z) + F_{n-2}(z) \qquad (E4)$$

$$F_1(z) = z; \quad F_0(z) = 1$$

for $z = x$ and $z = x+y$.

In particular, for calculating

$F_3(x+y)$

$$= (x+y)F_2(x+y) + F_1(x+y) \qquad (1)$$

$$= (x+y)(x^2+2xy+y^2+1) + (x+y) \qquad (2)$$

$$= (x^3+2x^2y+xy^2+x)+(x^2y+2xy^2+y^3+y) + (x+y) \qquad (3)$$

$$= x^3+2x^2y+3xy^2+2x+y^3+2y \qquad (4)$$

one can easily figure the "minimum" number of compares on sums: 1 for sorting $(x+y)$ in line 1, none to compute $x \cdot F_2$ and $y \cdot F_2$, 4 in line 3 for merging the two 4-term sums, 3 in line 3 for merging the final 2-term sum into the previous 6-term sum. That makes a total of 8. During the stepwise refinement of algorithm A3, measurement gave, successively: 20, 16, 14, 8.

For n=20, measurement gave for

$F_{20}(x)$ 264 initially, 142 in the end;
$F_{20}(x+y)$ 8110 initially, 2022 in the end.

Time of course, cannot be reduced to the same extent. In order to get a lead on what one may expect in terms of growth, let us consider the number of terms of F_n. For n even, let

$G(z)$ the number of terms generated intermediately for $F_n(z)$;
$R(z)$ the number of terms in the final result for $F_n(z)$.

It es easy to show that

$$G(x) = R(x) = n + (n/2)^2$$

$$G(x+y) = 2(n/2)^3 + (7/2)(n/2)^2 + (3/2)(n/2)$$

$$R(x+y) = (2/3)(n/2)^3 + (5/2)(n/2)^2 + (17/6)(n/2)$$

Thus

$$\frac{G(x+y)}{G(x)} = n - 1/2 + O(1/n)$$

$$\frac{R(x+y)}{R(x)} = (1/3)n + 7/6 + O(1/n)$$

Figure F1 shows these two curves and relates them to measurement curves

$$\frac{\text{time to compute } F_n(x+y)}{\text{time to compute } F_n(x)}$$

(a) at the outset of refining algorithm A3
(b) at the end of refining algorithm A3.

We observe

(1) Tuning brings down the entire time ration curve more or less evenly, and thus takes effect over the whole range. The slope does not change significantly.
(2) The time ratio is bounded by the two term ratio curves. For small n, its slope approaches the slope of $R(x+y)/R(x)$; for large n, it approaches the slope of $G(x+y)/G(x)$. That is, asymptotically the work involved grows with the number of terms that have to be generated to find the final result.

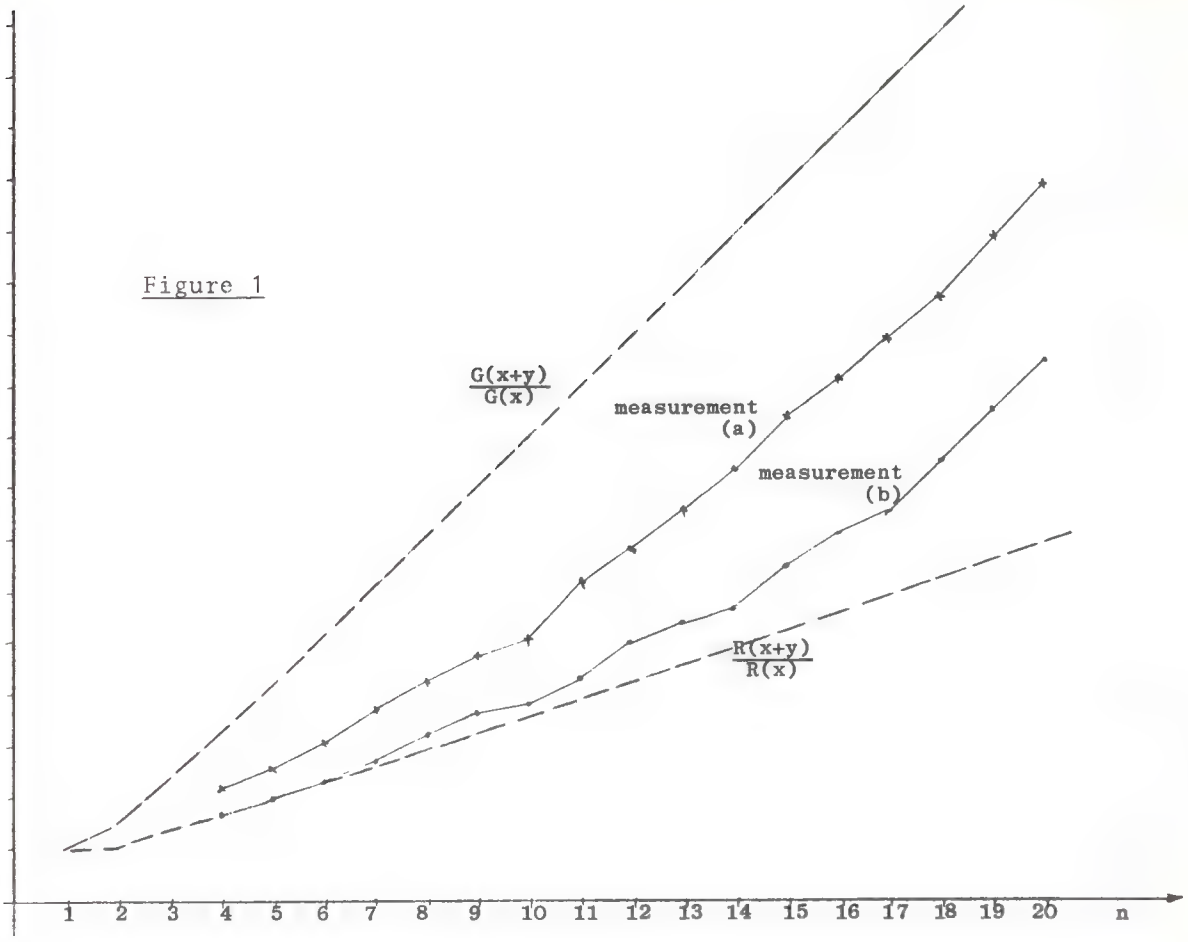


Figure 1

5. A Short Note on Powers of Sums

The computation of powers of polynomials has been discussed in a number of papers [e.g. 5,6,11,14,15]. Our main interest here is in a paper by Fateman [5].

For expanding sums raised to a positive integral power FORMAC has been using an algorithm that resembles NOMB [5]. This is not a bad choice with respect to cost as defined by Fateman. The cost of calculating the multinomial coefficients is kept to a minimum by computing only the different ones in a special purpose routine. Unfortunately, FORMAC used to first generate all terms, then simplify and sort them. When the algorithm is applied to polynomials, the space required is prohibitive. This unpleasant feature has been removed, so that the system now behaves more reasonable.

In expanding polynomials of the type $(x+x^2+\ldots+x^r)^n$, with or without integer coefficients, the space reduction should be and actually is proportional to a reduction from size binom(r+n-1,n) to size n(r-1)+1. For instance, for r=5, n=7, the experienced savings were about 30K in space and about 20% in time. With coefficients, the program takes about 65% longer than without coefficients. In either case, to expand a dense polynomial takes about 70% of the time needed to expand the corresponding sparse polynomial $(a_1+a_2+\ldots+a_r)^n$.

An aspect not considered by Fateman (intentionally) is sorting. Sorting still consumes a significant part of the total time. A drawback of the NOMB algorithm is that it takes absolutely no advantage of the given ordering of terms. We can probably profit by that information, if we use one of the binomial expansion algorithms (BINA or BINB). But they have recourse to multiplication of sums; NOMB does not. So, as long as FORMAC had a poor multiplication routine, we experienced really bad times (about double) with BINA and sparse polynomials. Things look better now. At the time of this report, not enough experiments have been carried out to make a satisfactory comparative statement. The matter will be taken up in a follow-up paper.

In the calculation of powers of sums, as in other areas, we found again that separating sorting considerations from simplification proper in the system routines was important. To a great extent it was this decision that made it possible to achieve a further improvement in the overall performance of FORMAC.

References

1. K. Bahr: Toward a Revision of FORMAC. SIGSAM Bulletin no. 29, Feb. 1974
2. K. Bahr, J. Smit: Tuning an Algebraic Manipulation System through Measurements, Proc EUROSAM, SIGSAM Bull., Aug. 1974
3. K. Bahr: Utilizing the FORMAC Novelties
Proc SEAS Ann Meeting, Zurich, Sept. 1974, pp 236-245
4. R.E. Collins: The SAC-1 System: An Introduction and Survey
Proc 2nd Symp on Symb and Alg Manipulation (S.R. Petrick, ed.) Los Angeles, March 1971, pp 144-152
5. R.J. Fateman: On the Computation of Powers of Sparse Polynomials. Studies Appl Math 53 (June 1974), pp 145-155
6. R.J. Fateman: Polynomial Multiplication, Powers and Asymptotic Analysis: Some Comments. SIAM J Comput 3 (Sept 1974) pp 196 -213
7. J. Fitch, A. Norman: An Idea for Comparing Data Structures
SIGSAM Bulletin no. 27,Sept.1973
8. J. Fitch: CAMAL User's Manual
University of Cambridge, England,1975
9. J. Fitch: CAMAL Data Structures
Technische Hogeschool Twente, memo no. 106, Nov. 1975
10. J. Fitch: Measuring CAMAL
Joint Meeting NIGSAM/SEAS SMC, London, Jan. 1975
11. W.M. Gentleman: Optimal Multiplication Chains for Computing a Power of a Symbolic Polynomial
Math of Comp 26(Oct. 1972),pp 935-939
12. A.D. Hall: The ALTRAN System for Rational Function Manipulation: A Survey.
Proc 2nd Symp on Symb and Alg Manipulation (S.R. Petrick, ed.), Los Angeles, March 1971, pp 153-157
13. A.C. Hearn: REDUCE 2 User's Manual
Stanford memo no. AIM-133, Oct. 1970
14. L.E. Heindel: Computation of Powers of Multivariate Polynomials
Over the Integers. J Comp and Syst Sci 6(1971) pp 1-8
15. E. Horowitz: The Efficient Calculation of Powers of Polynomials
J Comp and Syst Sci 7(1973) pp 469-480
16. F.K. Hwang, S. Lin: A Simple Algorithm for Merging two Disjoint Linearly Ordered Sets. SIAM J Comput 1(March 1972), pp 31-39
17. S.C. Johnson: Sparse Polynomial Arithmetic
Proc EUROSAM, SIGSAM Bull., Aug. 1974
18. D.E. Knuth: The Art of Computer Programming, vol 2 and 3
Addison-Wesley, 1969 and 1973
19. P. Marks: Design and Data Structure: FORMAC Organization in Retrospect.
Proc 1968 Summer Inst on Symb Math Comp (R. Tobey, ed.), IBM Boston Programming Center, June 1969
20. J. Smit, private communication, 1974
21. R. Tobey et al.: PL/I-FORMAC Symbolic Mathematics Interpreter
Contrib Pgm Lib 360D-03.3.004, September 1969
22. J. Xenakis: The PL/I-FORMAC Interpreter
Proc 2nd Symp on Symb and Alg Manipulation (S.R. Petrick, ed.) Los Angeles, March 1971, pp 105-114

Univariate Power Series Expansions in Algebraic Manipulation

by

Richard E. Zippel

Laboratory for Computer Science, MIT
(formerly Project MAC)

Abstract

In this paper we present a complete algorithm for the determination of univariate power series expansions of meromorphic functions on a Riemann surface. The difficulties involved when expanding at singularities of various forms are discussed. We demonstrate how to use these techniques to calculate limits and as an aid in solving polynomial equations. Finally we discuss several of the implementations of power series manipulation systems with special emphasis on the implementation in MACSYMA.

Introduction

There have been a large number of papers in the recent past which have incidentally mentioned power series computation [16, 2], but for the most part the expansions have been developed as a means to a particular end. More often than not the expansions where used to obtain the values of a function in a certain neighborhood. Consequently most of these papers were concerned with power series as a tool of numerical analysis. In this paper we are mainly concerned with the problem of obtaining the power series expansions. Most of the results contained here though not new, seem to have never been collected together in a useful manner for implementors of algebraic manipulation systems. We have discovered several applications of power series to algebraic manipulation and we suspect there are many more.

These applications all require an extensively developed series expansion system. The more functions such a system can handle in a "reasonable" manner the more powerful the algorithms would be. Our ultimate goal is to be able to find power series expansions for as many functions as possible. Here we present a detailed explanation of the techniques used by the sophisticated power series expansion system embedded in the MACSYMA system [13], a large mathematical system under development at MIT's Laboratory for Computer Science.

Given a function, $f(z)$ and a point α, our problem is to find a power series expansion of $f(z)$, if it exists, with the exponents restricted to some set $\mathfrak{S}$:

$$f(z) = \sum_{i \in \mathfrak{S}} a_i (z - \alpha)^i$$

We are interested in only the first few terms. So given a positive integer n we want to determine all the a_i such that $i < n$. If $\mathfrak{S}$ is the positive integers then we have what is known as the Taylor series expansion of $f(z)$. The Taylor series is the most common form of power series expansion and the easiest to compute, but alone this type of expansion does not cover a very broad range of functions (e.g. $\cot z$, $\log \sin z$ do not possess Taylor series at the origin), nor do they permit certain operations (e.g. quotients and radicals at certain places). By permitting $\mathfrak{S}$ to be the positive and negative integers we have what is known as a Laurent series. Essentially this extension permits division. A further extension, by letting $\mathfrak{S}$ be the rational numbers permits algebraic operations. While an appropriate substitution can usually be made to transform any computation involving one to another of these forms, the transformation cannot always be determined *a priori*. The meaning of the result will usually be clearer (especially in asymptotic analyses) if the expansion is expressed in terms of the variable (or variables) of expansion. The MACSYMA power series package permits all of these types of expansions.

In a forthcoming paper [21], we discuss multivariate expansions. The difficulties encountered are far more complex than those of univariate expansions and require a much more sophisticated manner of attack. However, many of the results mentioned here are useful in the multivariate case, while the multivariate case gives a bit more insight into the geometric aspects of power series expansions. It is hoped that these two papers will complement each other.

1.1. Basic Complex Variable Theory

In this section we will attempt to lay the ground work for the analytic portion of this paper concerned with functions of a single complex variable. All of the material is classical and is usually included in a good course in complex analysis. It is included to fix the terminology. A good classical text which covers much the same ground in a similar spirit is [5].

1.1.1. Properties of Functions

We denote the field of complex numbers by $\mathbb{C}$. If $f:\mathfrak{A} \rightarrow \mathfrak{B}$ associates elements of $\mathfrak{B}$ with elements of $\mathfrak{A}$ then f is called a *function*. There is substantial ambiguity as to whether or not a function is single valued. This ambiguity will be eliminated by making the domain of definition of all functions a connected Riemann surface. We let $\mathbb{C}$ have the usual topology. By a neighborhood of α we will understand an open set which contains α. A punctured disk centered at α is a neighborhood of α with α removed.

A function is called *analytic* or *holomorphic* in a region if it satisfies the Cauchy-Riemann equations within that region. That is, if:

$$f(z) = w$$

$$w = u + iv \quad \text{and} \quad z = x + iy$$

the Cauchy Riemann equations for f are

$$\frac{\partial u}{\partial x} = \frac{\partial v}{\partial y} \quad \text{and} \quad \frac{\partial v}{\partial x} = -\frac{\partial u}{\partial y}.$$

Notice that functions are analytic in regions and need not be analytic over the entire complex plane. The property of analyticity for a function of a complex variable corresponds to that of differentiability for a function of a real variable.

1.1.2. Singularities

A function which assumes but one value for each value of its argument will be called *uniform*. A "function" which has more than one value is called *multiform*. We say that a function is bounded in a neighborhood if the image of the neighborhood attains a maximum absolute value. Otherwise we say the function is unbounded in that neighborhood.

Of the values a uniform function may take on, two are of particular importance. If the function $f(z)$ is unbounded in every neighborhood of a point, α, we say that $f(z)$ has an *infinity* at α. If $f(\alpha)$ is equal to 0 then we say that α is a *zero* of $f(z)$. Notice that the function $e^{1/z}$ does have an infinity at the origin, but there exist sequences which have zero as their accumulation point and whose image is bounded. By approaching the origin along the imaginary axis the absolute value remains constant.

By the expression:

$$\lim_{z \rightarrow \alpha} f(z) = R \qquad (2)$$

we mean that the limits of $f(z)$ along all paths to α are the same and they are equal to R. If the limits are different then we say that the limit of the function at that point is *indeterminate*.

If a function is uniform and analytic in a disc punctured at a point α and at α the function is not analytic, the point is called an *isolated singularity*. Isolated singularities may be divided into three classes, *removable singularities*, *poles* and *essential singularities*. If the R in *(2)* is finite and necessarily different from $f(\alpha)$ then α is called an removable singularity. The singularity could be "removed" by defining $f(\alpha)$ to be R. Let $f(z)$ have an isolated singularity at α which is not removable. Then $z = \alpha$ is a pole of $f(z)$ if

$$\lim_{z \rightarrow \alpha} (z - \alpha)^n f(z) = 0 \qquad (3)$$

for some positive value of n. It is clear from *(3)* that a pole is also an infinity. In fact $1/f(z)$ is analytic and bounded in some neighborhood of a pole of $f(z)$ including the pole of $f(z)$.

If an isolated singularity is not a pole or removable singularity it must be an essential singularity. To distinguish an essential singularity from a pole it is necessary to determine whether or not the limit of the function at the singular point is determinate. Thus we merely examine the reciprocal of the function. If it has the point as an ordinary point, then the point is a pole of the function. Otherwise the point is an essential singularity. A beautiful theorem due to Weierstrass points to the richness of essential singularities [19, t ii, pp 122-124], [18]. *In the immediate vicinity of an isolated essential singularity of a uniform function, the value of the function is arbitrarily close to any preassigned value.*

1.1.3. Riemann Surfaces.

A multiform function may have a variety of different values at any one point. These values may be placed into classes in a such a manner that that the multiform function may be thought of as several uniform functions. Consider the following example, $f(z) = \sqrt{z}$. The function $f(z)$ may be thought of as being composed of two uniform functions, analytic in a punctured disk at the origin.

$$f_1(z) = \sqrt{|z|}\, e^{i(\arg z)/2}$$

$$f_2(z) = \sqrt{|z|}\, e^{i(2\pi + \arg z)/2}.$$

Each of these functions corresponds to a *branch* of the function. A point where two or more branches coalesce is called a *branch point* of the function. Notice that $f(z)$ has two branch points, the origin and ∞. If z varies along a circle which encloses but one of the branch points, then $f(z)$ will alternate between $f_1(z)$ and $f_2(z)$. We can say that $f(z)$ is "analytic" on this doubled copy of of the complex plane. This surface is known as the *Riemann surface* of $f(z)$. Intuitively this can be thought of as two rubber discs, their center being the origin, and their edges, the point ∞. They are connected at the branch points, 0 and ∞. If they are sliced along a line running from one branch point to the other, and the edges cross connected (impossible in the physical world), we have the proper structure. This slice is called a branch cut.

Univariate Expansions

The obvious technique to obtain a univariate power series expansion, an algorithm which is often used in algebric manipulation system demonstrations, is Tavlor's formula:

$$f(z) = \sum_{i=0}^{\infty} \left. \frac{f^{(k)}(z)}{k!} \right|_{z=\alpha} (z-\alpha)^k \qquad (4)$$

This formula can lead to disastrous "intermediate expression swell" (algebraic manipulation's equivalent to round-off error). A more severe problem is that $f(z)$ must be analytic in a neighborhood of α for the evaluation to be possible. Thus expansions at removable singularities still cannot be obtained. By taking limits instead of evaluating $f^{(k)}(z)$ this restriction is removed. Poles still cannot be handled since the derivatives do not take on finite values.

Laurent's expansion formula allows functions which possess poles:

$$f(z) = \sum_{k > -\infty} \left[\frac{(z-\alpha)^k}{2\pi i} \oint_{\mathfrak{C}} \frac{f(w)\,dw}{(w-\alpha)^{k+1}} \right] \qquad (5)$$

where $\mathfrak{C}$ is a circle around α. In itself this is an extremely elegant formula, but it is often very difficult to evaluate the integrals, far tougher than the derivatives in (4).

Note that if α is finite we can expand the function $f(w+\alpha)$ at the origin and then replace w by $z-\alpha$ to get the proper power series expansion at α. Similarly any point on the projective plane can be transformed to the origin by a bilinear transformation. This transformation can be applied to any number of independent variables. Thus from now on we will always assume the expansion is to be taken at the origin.

2.1. Algorithmic Techniques

Since all our power series expansions will be finite we will need some method to describe number of terms of the expansion computed. In general a missing term in a power series indicates that the associated coefficient is zero. If we have a power series representation of the function $f(z)$:

$$a_\alpha z^\alpha + a_\beta z^\beta + \ldots \qquad (6)$$

for which all the terms $a_\kappa z^\kappa$ with $\kappa \leq n$ are correct the we say that the degree of (6) is n. The following power series' for e^z both have degree 2.

$$1 + z + \frac{z^2}{2} + \ldots$$

$$1 + z + \frac{z^2}{2} + \frac{z^3}{3} + \ldots$$

If a function takes on a finite, non-zero, value at a point we say it has *order* zero. We write $ord_p\ e^z = 0$ or $ord\ f(z)$ when the point is given by context. In general if $ord\ f(z) = n$ then the first non-zero term in the power series expansion of $f(z)$ has exponent n. When $f(z)$ does not possess a power series expansion its order is undefined. If n is greater than zero then $f(z)$ has a zero of order n. If n is less than zero then $f(z)$ has a pole of order n.

The method we present here is not very deep. Anyone who has calculated a few power series by hand has probably seen it, but there are a few wrinkles. If we want the expansion for $f(z) = \sin z\, e^z$ at the origin valid to degree z^3, then by multiplying the expansions for $\sin z$ and e^z

$$f(z) = \left(z - \frac{z^3}{6} + \ldots\right)\left(1 + z + \frac{z^2}{2} + \frac{z^3}{6} + \ldots\right)$$

$$= z + z^2 + \frac{z^3}{3} + \ldots$$

The first thing to notice is that we really don't need the z^3 term of e^z. It is possible to have the reverse of this behavior occur. We may be in need of more terms than first thought, rather than have a surplus. For instance consider $(e^z - 1)/z$. When expanding at the origin to degree 3.

$$f(z) = \left[z + \frac{z^2}{2} + \frac{z^3}{6} + \ldots\right]/z$$

Now we need another term from e^z, the z^4 term, in order to get the z^3 term correct in answer. The resolution of this problem is stated in Theorem 1.

Theorem 1: If $f_1, f_2, \ldots, f_m$ are functions which possess power series expansions, and the order of f_i is o_i, with $\Sigma_i\ o_i = l$, and we wish the expansion of $f_1 f_2 f_3 \ldots f_m$ good to degree n then f_i must be expanded to degree $n - l + o_i$.

Proof: Let $f_i = a_{i\ o_i} z^{o_i} + \ldots$. Then $f_i\ z^{-o_i} = a_{i\ o_i} + \ldots$ so if each of the $f_i\ z^{-o_i}$ is expanded to degree p and the results multiplied together, we get a power series whose degree is p. So we now have $z^{-l}[\Pi_i\ f_i]$, or $\Pi_i\ f_i$ valid to $p + l$. We want $p = n - l$, so each f_i must be expanded to degree $n - l + o_i$.

As a corollary we have a similar result for exponentiation:

Corollary: For the expansion of f^s to be valid to degree n, the expansion of f from which f^s was derived must be valid to degree $n - (s-1)\ ord\ f$.

Proof: In Theorem 1, let $l = s\ ord\ f$, $o_i = ord\ f$.

Note that though the proof is not valid for non-integral s, the corollary is. A proof for non-integral s is easily constructed along the lines of Theorem 1.

2.1.1. Exponentiation of power series

While it is clear that exponentiation to integral powers can be done by repeated multiplication, or squaring, a superior method can be devised which is valid even for non-integers. If

$$f(z) = \sum_{n \geq 0} a_n z^n$$

and

$$f(z)^s = \sum_{n \geq 0} b_n z^n$$

then

$$b_0 = a_0^s$$

$$\sum_{0 \le k < n} [k(s+1) - n]\, a_k\, b_{n-k} = 0 \qquad (7)$$

Formula *(7)* was known to Euler [3] although it may be even older. A quick derivation of *(7)* may be arrived at by differentiation [11]. An elegant proof using properties of double sums is due to von Holdt [9], though we shall not repeat it.

Notice that this formula, along with most other power series formulae, requires the expansion to be of order 0, and all the exponents to be integers. A power series of this form will be termed in *normal form*:

$$f(z) = a_0 + a_1 z + a_2 z^2 + a_3 z^3 + \ldots$$

If this is not the case,

$$f(z) = a_l\, z^l + a_m\, z^m + a_n\, z^n + \ldots$$

then the formula is applied to $f(z)\ z^{-l}$ and the answer is readjusted in an appropriate manner. This is done in the following manner for exponentiation:

$$f(z)^s = [f(z)\ z^{-l}\ z^l]^s = z^{ls}\ [f(z)\ z^{-l}]^s$$

2.1.2. Composition of Functions

The next most complicated operation is composition of functions. For instance consider, $e^{\sin z}$, to degree 3. First we expand $\sin z$:

$$\sin z = z - \frac{z^3}{6} + \ldots \qquad (8)$$

The obvious step, is to substitute *(8)* into *(9)*:

$$e^z = 1 + z + \frac{z^2}{2} + \frac{z^3}{6} + \ldots \qquad (9)$$

But again there is a much better way. Let

$$f(z) = \Sigma_n\, a_n\, z^n \quad \text{and} \quad e^{f(z)} = \Sigma_n\, c_n\, z^n$$

where both series are normal. We know that

$$D_z\, e^w = w'\, e^w \qquad (10)$$

So by replacing w in *(10)* by *(9)*:

$$\sum_{n \ge 0} c_{n+1}\,(n+1)\, z^n = \Big[\sum_{n \ge 0} (n+1)\, a_{n+1} z^n\Big] \Big[\sum_{n \ge 0} c_n\, z^n\Big] = \sum_{n \ge 0} z^n \sum_{k=0}^{n} (k+1)\, a_{k+1}\, c_{n-k}$$

so

$$c_{n+1} = \sum_{k=0}^{n} \frac{(k+1)}{(n+1)}\, a_{k+1}\, c_{n-k}$$

or

$$c_n = \sum_{k=1}^{n} \frac{k}{n}\, a_{k+1}\, c_{n-k}$$

naturally $c_0 = 1$. In our problem we get:

$$e^{\sin z} = 1 + z + \frac{z^2}{2} + \ldots$$

But what if we had wanted $e^{\cos z}$? Then a_0 would not have been zero as required by the formula. The key is to follow the technique used in exponentiation, we expand $e\ e^{\cos z - 1}$ instead.

What if there had been negative powers? Then the function would have had an essential singularity at the origin, as in $e^{1/z}$. Consequently the power series expansion, if there had been any, would have an infinite number of negative terms. Notice that this would have wreaked havoc on the formula in Theorem 1. For now we shall assume that there are no essential singularities at the point of the expansion. Later we shall describe a method of handling certain case of essential singularities.

Naturally a formula similar to the one just derived could be generated for every function which is the solution of a linear differential equation. A more reasonable technique would be to derive the formulae automatically from a differential equation for the function.

One problem we have glossed over here, is how to handle $f(g(z))$ when *ord* g is equal to zero. The basic technique is to let $g_1(z) = g(z) - g(0)$. We are then looking for a formula for $f(g(z))$ in terms of $f(g_1(z))$ and $f(g(0))$, both of which are easily expressed as a power series. If a function $\phi(u + v)$ can be expressed algebraically in terms of $\phi(u)$ and $\phi(v)$ then ϕ is said to have an algebraic addition formula. According to a theorem due to Weierstrass [17], for f to possess an algebraic addition formula $f(z)$ must be expressible as a rational function of z, $e^{i\pi z/\omega}$, $\wp(z)$ or $\wp'(z)$, where $\wp(z)$ is the Weierstrass-P function. So if a function is not a rational function it must be either periodic or doubly periodic to possess an addition formula. Thus we can expand $f(g(z))$ when $g(z)$ has order zero, *only* when f is of the above form. We know of no general technique for obtaining expansions of functions like $Y_n(\sin z)$ where Y_n is a Bessel function, although it is easy to see how a system which relies on pattern matching could handle many special cases.

2.1.3. Branch Points

We continue to consider only functions which are meromorphic on some Riemann surface. Thus far we have only consider simple points. The interesting part is what is left. We will divide branch points into classes: those at which an infinite number of sheets of a Riemann surface come together and those at which a finite number of sheets coalesce. If a function has an infinite number of sheets then its inverse is periodic. Since all periodic functions of one variable are algebraically dependent on the exponential function and the Weierstrass-P function, we divide the infinite sheeted case into two parts, *logarithmic* branch points and $\wp$-*ic* branch points. For the present we restrict ourselves to expansions at algebraic and logarithmic branch points.

Following the lead of our earlier results, the expansion of $f(z) = \sqrt{(z + z^2)}$ would proceed as follows

$$f(z) = z^{1/2}\sqrt{(1 + z)}$$

$$= z^{1/2}\left(1 + \frac{z}{2} - \frac{z^2}{8} + \frac{z^3}{16} + \ldots\right)$$

$$= z^{1/2} + \frac{z^{3/2}}{2} - \frac{z^{5/2}}{8} + \frac{z^{7/2}}{16} + \ldots$$

Notice that this expansion cannot be produced by Taylor's or Laurent's formulae. Both consider only integer exponents. It is obvious that we can manipulate such series, at least formally, like any other power series. When evaluating such a series care must be taken with the square roots to ensure that they are all taken along the same branch. For asymptotic analysis, power series with fractional exponents are quite reasonable; especially since the origin, being a branch point, is one of the places where some of the sheets coincide. A typical example of this type of analysis can be seen in [10, pp112-118].

We can formally manipulate logarithmic singularities in much the same way.

$$\log(\sin z) = \log\left(z - \frac{z^3}{6} + \ldots\right)$$

$$= \log z + \log\left(1 - \frac{z^2}{6} + \ldots\right)$$

$$= \log z - \frac{z^2}{6} + \ldots$$

This sort of series is even stranger, and there many surprising differences in how it must be handled from the usual power series. However it still retains the asymptotic nature of a power series.

2.1.4. Essential Singularities

In general a function possessing an essential singularity cannot be expressed as a power series with a finite number of negative order terms. However, there are certain expressions which represent functions without essential singularities (in some neighborhood) but which possess subexpressions which do have essential singularies (in the neighborhood). For example

$$\frac{e^{\csc z}}{e^{\cot z}} \qquad (11)$$

Both the numerator and the denominator possess essential singularities since both $\csc z$ and $\cot z$ have poles at the origin. By converting *(11)* to

$$e^{\csc z - \cot z}$$

we eliminate the pole in the exponents and easily get the following expansion.

$$1 + \frac{z}{2} + \frac{z^2}{8} + \frac{z^3}{16} + \frac{3z^4}{128} + \frac{37z^5}{3840} + \ldots$$

A similar sort of problem occurs with an expressions like

$$(e^{1/z})\sin z$$

Collapsing the exponents we get

$$e^{\sin z/z} = e\left(1 - \frac{z^2}{6} + \frac{z^4}{45} + \ldots\right)$$

These two techniques are quite powerful and will solve many problems with essential singularities which arise in practice.

There is another possible manner to handle essential singularities, a trick in much the same spirit as the trick used at logarithmic branch points. In the expansion of *(11)* we expand the numerator as

$$e^{z^{-1} - z/6 + 7z^3/720 + \ldots}$$

$$= e^{1/z}\left(1 + \frac{z}{6} + \frac{z^2}{72} + \frac{131\,z^3}{6480} + \ldots\right)$$

and the denominator similarly. When the two expressions are divided the $e^{1/z}$ terms factor out and we have the desired power series. But this technique can be extremely dangerous since the "essential singularity terms" will dominate a power series, unlike the logarithmic terms. So if they are truncated accidently the resulting series can be very misleading. Consider the "power series"

$$(1 + z\,e^{1/z})^{-1} = 1 - e^{1/z}\,z + e^{2/z}\,z^2 + \ldots$$

While formally the expansion is valid it converges outside the circle of radius λ and is useless in determining the local nature of the function. λ is absolute value of the zeroes of $e^{1/z}\,z - 1$.

2.2. Branch Points Reconsidered

There are two ways to study the behavior of functions at a branch point. In this section we will use the concepts of a Riemann surface and parametrization. These techniques are relatively intuitive and do not require very deep mathematics. The deeper, geometric view is necessary when multivariate expansions are considered. It is presented in [21].

2.2.1. Algebraic Branch Points

Consider the function $w = \sqrt{(z - z^2)}$. It has a two sheeted Riemann surface, $\mathfrak{R}$ and two branch points, at the origin and at $z = 1$. By considering the problem locally, in this case in an open neighborhood of the origin which does not include the point $z = 1$ we can "resolve" the singularity. The key idea is to "unwind" these two sheets (locally they are discs) by slicing one of the branch cuts, flattening the discs onto a plane and reconnecting the branch cut. Thus we have a mapping from an open neighborhood of the origin to $\mathfrak{R}$. This mapping is called a *local parametrization* of $\mathfrak{R}$. For each branch point on each Riemann surface there are an infinite number of local parametrizations. We seek one which maps a power of t, the *uniformizing parameter*, into z. For w we have $t^2 \to z$. Notice that as t goes around the origin once z circuits the origin twice, just as

desired. Thus there is a bijection from the *t-plane* to $\mathfrak{R}$ just as desired. In fact we can expand w as a Taylor series in t:

$$w = t\sqrt{(1 - t^2)}$$

$$= t\left(1 - \frac{t^2}{2} + \frac{t^4}{8} + \ldots\right)$$

$$= t - \frac{t^3}{2} + \frac{t^5}{8} + \ldots \qquad (12)$$

There are other expansions possible; however,

$$w = -t + \frac{t^3}{2} - \frac{t^5}{8} + \ldots \qquad (13)$$

For each choice of uniformizing parameter a power series expansion is induced. Notice that replacing t by -t in *(13)* we get *(12)*

Let x and y be related by the polynomial $f(x,y) = 0$. Let $t^n = x$ be a parametrization of f. Two power series expansions for y in t are called *equivalent* if by substituting ωt for t in one of them yields the other expansion and ω is an n-th root of unity. In this case n sheets of the Riemann surface will coalesce at the point of expansion. There are precisely m non-equivalent expansions at a point p if there are m places on the Riemann surface of f lying over p.

2.2.2. Logarithmic Branch Points

Again we seek a parametrization which will unwind the branch point at the origin. In this case the Riemann surface has an infinite number of sheets, but again the only branch points are at the origin and at ∞. In fact the Riemann surface can be thought of as the limit of the Riemann surfaces for $w^n = z$ as $n \to \infty$. These surfaces are parameterized by $z \to t^n$, or equivalently $z \to e^{n \log t}$. Because of the $\log t$ in this parametrization it will only cover a finite sheeted Riemann surface. The parametrization $z \to e^t$ will cover an infinite sheeted Riemann surface and is closer to what we need. But notice that if we were to use it alone, even polynomials would not be expressible as a power series.

Proposition: $\lim_{z \to 0} z \log^n z = 0$ for all real n.

The way out of this predicament is to parameterize the system in two variables, both t and z. We now have a power series in more than one variable, and in the most general case it is actually necessary to consider the expansion to be a multivariate power series and use the techniques of [21]. But frequently we can simply relegate t to the coefficient domain of the power series, and thus permit z to have coefficients which are rational functions in t. When this is done it is not necessary to find a truncation level for t.

It is only necessary to revert to multivariate power series when each function is to have a unique representation. Consider the following example:

$$\sqrt{(1 + \log \sin z)} =$$
$$\sqrt{(1 + \log z - \frac{z^2}{6} - \frac{z^4}{180} + \ldots)}$$

If the next step were to be taken (expansion of the square root) there would be expressions like $\sqrt{(1 + \log z)}$ in the power series. There is a significant difference between expressions like that and $\sqrt{3}$. In one case a particular value has been chosen for a constant while in the other we are picking a particular branch for a function to take. That is we are parameterizing the function again. This is deceptive since there is only a logarithmic branch point at the origin, there is no algebraic branch point. This is precisely the point when it becomes necessary to consider $t = \log z$ as another expansion variable.

Unfortunately $\log z$ is not always sufficient. If we have a power series in t (= $\log z$) the same types of expressions which force a $\log z$ can force a $\log t$. Thus we are left with an infinite tower of variables z, $\log z$, $\log \log z$, $\log \log \log z$, . . . of which we might need any of them. Needless to say it is rare when even $\log z$ is necessary and far rarer when any of the others are.

Applications

3.1. Limits

In a recent paper [12], J-P. Laurent discusses a program which calculates limits by heuristics. What he appears to do is to attempt to calculate the Taylor series expansion of the function at the limit point and use the leading term of the expansion to indicate the limit. When he fails he manipulates the expression (in a manner based on heuristics) and attempts the expansion again. He gives a large number of examples two of which he is unable to evaluate. Since our power series expansion techniques are so much more powerful than Laurent's it is interesting to see what sort of problems our technique can handle and which ones are beyond its capabilities.

First recall the basic ideas behind using power series expansions to obtain limits. First assume that we wish the limit of $f(z)$ as z approaches α. For there to be a power series expansion of $f(z)$ at α, f must be continuous. Thus we are considering only two sided limits. If $\alpha \in \mathbb{C}$, the we can transform α to the origin by a linear transformation. If α is equal to infinity we may use the bilinear transformation $z \to 1/z$ to move α to the origin. We may assume $\alpha = 0$. If $f(z)$ has the power series expansion

$$f(z) = a_\alpha z^\alpha + a_\beta z^\beta + \ldots$$

then the limit of $f(z)$ at zero is (1) zero if α is greater than zero, (2) a_α if α is equal to zero and (3) infinity if α is less than zero.[1]

From now on we assume that α is zero. By the proposition of the last chapter we can permit the higher order terms to involve $\log z$, $\log \log z$ and so on. The presence of these terms in the

1. Determination of the sign of infinity (if the function is assumed to be of a real variable) may be made by considering the sign of a_α.

coefficients does not affect the value of the limit. These terms still tend to zero with z.

Consider the problem: $2\,z^{\sin z} - z^{\tan z}$. Our algorithm immediately transforms the function into:

$$2\,e^{(\log z)(\sin z)} - e^{(\log z)(\tan z)}$$

The power series expansion of the exponents both have order 1 so they are plugged into the formula for e^z. The result is

$$1 + (\log z)\,z + \frac{(\log^2 z)\,z^2}{2} + \ldots$$

By permitting the use of logarithms in the coefficient domain we obviate some of the complicated transformations to which Laurent is forced to resort. Consider the problem

$$\frac{\log \sin z}{\log \tan z}.$$

Laurent's program could not handle this problem since it was unable to transform the expression into anything usable. He suggests that by expanding the insides of the logarithms and then applying his transformations the answer could be deduced. In our case it is a straight forward expansion. The numerator is:

$$\log z - \frac{x^2}{6} - \frac{x^4}{180} + \ldots$$

and the denominator

$$\log z + \frac{z^2}{3} + \frac{7z^4}{90} + \ldots$$

Taking the ratio

$$1 - \frac{z^2}{2\log z} - \frac{(\log z - z)\,z^4}{12\log^2 z} + \ldots$$

and the limit is 1.

We have numerous advantages over Laurent in that our expansion routines are so powerful. This is a prime situation where the *desiderata* of permitting as many functions as possible paid off handsomely. One of the major advantages of our system is that it is rather easy to see precisely what information needs to be added in order to incorporate new functions.[2] Because of this modularity a system which relies on power series manipulation can be extended to handle increasingly complex special functions merely by extending the power of the series expansion routines. Since the information needed by the series expansion routines is so small, (expansion at the origin and addition formula or translation formula) it is easy to add special functions.

3.2. Solutions of Polynomial Equations

When trying to solve a polynomial equation in one variable in terms of radicals it is useful to be able to determine if a polynomial differs from a perfect power of a polynomial by a constant. This greatly decreases the degree of the polynomial which must be solved. Since there do not seem to be any algebraic manipulation systems which can solve irreducible polynomials of degree greater than four (excepting cyclotomic polynomials and a few other cases) this improves matters greatly. If we are trying to solve $p(x) = 0$ and we note that $p(x)$ is equal to $q(x)^n - c$, then we merely solve $q(x) = c^{(1/n)}$ for each of the nth roots of c. We present a rather unusual use of Taylor series expansion to find $q(x)$ when $p(x)$ and n are given.

First a notice that if we take the nth root a polynomial (of degree d) as a power series, the fact that all the terms of degree d/n to $d + d/n$ are zero implies all the higher order terms are also zero. Notice that the convolution in *(7)* decouples with that many zero terms. The basic idea is to replace the constant term by a new constant, c. After taking the nth root the resulting Taylor series will have a polynomial coefficients. For the polynomial to be a perfect nth power all of the polynomials must have a common solution. This solution could be obtained by taking gcd's of all the coefficients but there is a better way. If

$$a_d x^d + a_{d-1}x^{d-1} + \ldots + a_0$$

is a perfect power then $a_0 x^d + \ldots + a_d$ is also. What is more important is that the first coefficient which involves c is the dth term in the nth root and what is better is that it is linear. Consequently to check whether the polynomial is a perfect nth power we must determine that, (1) the terms of the nth root of degree d/n to d-1 must all be zero, (2) the value of c which makes the dth coefficient zero must be an integer and (3) the value of c which satisfies (2) must make the next d/n coefficients zero. This procedure is very fast since there is only one place where a new variable is introduced, the calculation of the dth term, and after that the deduced value is used to compute the higher terms. Also it will determine that the polynomial cannot be an nth power very quickly because of condition (2).

Consider the following example, we wish to solve the following irreducible polynomial:

$$343\,x^9 - 147\,x^7 + 294\,x^6 + 21\,x^5 - 84\,x^4 + 83\,x^3 + 6\,x^2 - 12\,x + 10$$

Since the greatest common divisor of 147 and 9 is 3, this polynomial (up to an additive constant) is either a perfect cube or it is not a perfect power. So we take n to be 3. Reversing the coefficients and replacing the high order coefficient by c we get:

$$c\,y^9 - 12\,y^8 + 6\,y^7 + 83\,y^6 - 84\,y^5 + 21\,y^4 + 294\,y^3 - 147\,y^2 - 343$$

Viewing this as a power series, its cube root is

$$7 - y^2 + 2\,y^3 + \frac{(c-8)\,y^9}{147} + \frac{(2c-16)\,y^{11}}{1029} - \frac{(4c-32)\,y^{12}}{1029} + \ldots$$

2. We ignore the problem of higher order essential singularities. This is probably one of the most difficult of all the problems in univariate expansions, the complete incorporation of the higher class essential singularities in a uniform manner in a power series system.

This means c must be equal to 8 and we get:

$$(7x^3 - x + 2)^3 + 2$$

and must find the zeroes of

$$7x^3 - x + 2 - \omega$$

where ω runs through the cube roots of -2.

Implementation

Over the past few years there have been a number of algebraic manipulation systems which could manipulate power series, however in the last three years there has been a flurry of activity, with virtually every system which is still being actively developed, adding a power series manipulation system of some sort. Of the systems which have been developed there are a large number of different ways in which the system is embedded. All of these systems already have large amounts of "formula crunching" power available, the crucial difference being how they utilize that power. The most straight forward implementation is that in Altran [1]

4.1. Altran system

The basic idea is that any function which can be expanded in a univariate power series can be transformed, relatively easily, into one which can be expanded as a normal power series. Since all power series are going to be normal the natural way to represent them is as arrays, which is just what was done. The elements of the arrays are rational functions. The system maintains the order of truncation of each power series ("TPS" in the ALTRAN terminology) internally so the user need not be bothered.

Maintaining the order of expansion to the maximum the manipulations will permit is merely the reverse problem to that of generating a power series to a particular order. Theorem 1 of Chapter 1 and its Corollary can be used. So if A and B are TPS's and we let $tlvl\ A$ represent the truncation level of A. Then

$$tlvl\ AB = \min(tlvl\ A + ord\ B,\ tlvl\ B + ord\ A)$$

$$tlvl\ A^n = tlvl\ A + (n - 1)\ ord\ A$$

To manipulate a TPS, there are a large set of routines which must be explicitly called, with the exception of addition and multiplication by a constant. There are no special routines to manipulate multivariate power series, rather the manual suggests utilizing the mapping technique described in [21].

4.2. SCRATCHPAD System

The SCRATCHPAD implementation [15] is very interesting in that it manipulates the recurrence relations of the coefficients instead of the coefficients themselves. By assuming the expansions do not contain any terms of negative degree, branch points or essential singularities, the recurrence relations are precisely those used by the ALTRAN scheme. An advantage of this technique is that no actual computation is performed until particular coefficients are required. After the recurrence relations have been determined, they could be easily optimized and translated into a program if a large number of terms are desired. And finally, if the user is not sure of how many terms he will need, he can set up the recurrences, compute a few terms, decide how many he really will want, and compute the additional terms without recomputing the earlier ones.

The SCRATCHPAD package is implemented by means of a translational mechanism. Thus the expression $sin\ z$ is translated:

$$sin\ z \rightarrow s_i$$

where s_i are the coefficients in the expansion of $sin\ z$. If a and b were power series with

$$a = \Sigma_i\ a_i z^i$$

$$b = \Sigma_i\ b_i z^i$$

then the following transformations would be made.

$$a + b \rightarrow a_i + b_i$$

$$a\ b \rightarrow \Sigma_j\ a_j\ b_{i-j}$$

One particularly nice feature of this scheme is that the translations could probably be modified to handle any sort of expansion, Bessel functions, fourier series, q-series. This sort of extension has not been achieved in any system, including SCRATCHPAD, but it is a useful feature pointed out by Gentleman [7].

4.3. MACSYMA System

Truncated power series have been implemented in MACSYMA [13] in two manners. Both implementations permit expansions at branch points, poles and certain removable essential singularities. The user specifies a particular degree for the expansion and the system then decides to what degree the subexpressions must be expanded. The difference between the two packages is that one permits the power series to be extended after it has been created. These "extendable" power series may be manipulated just like the other "non-extendable" variety, but at any point in a computation any number of additional terms may be calculated without recalculating the lower degree ones.

4.3.1. MACSYMA Examples

In this section we present a few examples of the use of the power series expansion package in MACSYMA partly to reveal its power and ease of use, and partly to demonstrate the use of the techniques outlined in previous chapters. In its simplest form a truncated power series may be obtained by the following incantation.

```
(C1) TAYLOR(SIN(X)/X,X,0,8);
                  2    4     6      8
                 X    X     X      X
(D1)        1 - -- + --- - ---- + ------ + . . .
                 6   120   5040   362880
```

Here we requested the power series expansion of

sin x/x at the origin, up to and including the terms of degree 8. In the next example we get the binomial formula. The beautiful form of the coefficients is due to the "factored form" representation of the coefficients. The exponentiation formula, *(7)* was used to produce the result though.

```
(C2) TAYLOR((1+X)**N,X,0,5);
                                  2                    3
                     (N - 1) N X    (N - 2) (N - 1) N X
(D2)     1 + N X + ------------ + --------------------
                         2                  6

                                  4
   (N - 3) (N - 2) (N - 1) N X
 + ---------------------------
                 24

                                          5
   (N - 4) (N - 3) (N - 2) (N - 1) N X
 + ---------------------------------- + . . .
                    120
```

In the next example, there is an essential singularity in both the numerator and the denominator. The singularity is removed by combining the exponents.

```
(C3) TAYLOR(%E**CSC(X)/%E**(1/X),X,0,5);
                  2        3        4          5
             X   X    131 X    509 X    15163 X
(D3)     1 + - + -- + ------ + ------ + -------- + . . .
             6   72    6480    155520   6531840
```

The following is a multivariate expansion calculated by the natural technique. Note how much more complicated the answer is than in D5 where the mapping technique was used. Part of the simplicity is due to the factored form representation used in MACSYMA.

```
(C4) TAYLOR((1+X+Y)**N,X,0,2,Y,0,2);
                                  2
                     (N - 1) N Y
(D4)     1 + N Y + ------------ + (N + (N - 1) N Y
                         2

     2                 2
   (N  - 3 N + 2) N Y                 (N - 1) N
 + ------------------- + . . .) X + (---------
            2                              2

                                 2                      2
   (N - 2) (N - 1) N Y   (N  - 5 N + 6) (N - 1) N Y
 + ------------------- + ---------------------------
            2                          4

            2
 + . . .) X  + . . .

(C5) TAYLOR((1+X+Y)**N,[X,Y],0,3);
                                          2
                          (N - 1) (X + Y)  N
(D5)     1 + (X + Y) N + ------------------
                                 2

                                                3
                           (N - 2) (N - 1) (X + Y)  N
                         + ------------------------- + ...
                                       6
```

The extendable power series feature is perhaps the most exciting portion of the MACSYMA implementation. It provides the user with the naturalness of expression of the standard MACSYMA power series package but does not lock into a specific order of expansion. Though inspired by the SCRATCHPAD power series expansion package, MACSYMA's is built upon completely different principles.

Below we present a MACSYMA solution to "problem 8" of the *SIGSAM Bulletin*. When originally solved [20], the author used an older version of the current system which did not provide the extendable techniques. The problem reduces to finding a few terms of the expansion of the generic solution of a second order difference equation. The following dialogue is an explicit calculation of that solution.

```
(C6) B:TAYLOR((1-Q-Q*U)*%E**(-T*(1+U))+
              (1-Q+Q*U)*%E**(T*(1-U)), T,0,1);
(D6)          2 - 2 Q + (4 Q - 2) U T + . . .

(C7) C:TAYLOR((1-2*Q)*%E**(-2*T*U),T,0,1);
(D7)          1 - 2 Q + (4 Q - 2) U T + . . .

(C8) H1:TAYLOR(((1+U)*%E**(T*(1-U))+
                (1-U)*%E**(-T*(1+U)))/2, T,0,1);
(D8)                          1

(C9) DISC:SQRT(B**2-4*C);
(D9)           2 Q + (- 4 Q + 2) U T + . . .

(C10) X:(B+DISC)/2;
(D10)                         1

(C11) Y:(B-DISC)/2;
(D11)          1 - 2 Q + (4 Q - 2) U T + . . .

(C12) ANS:((H1-Y)/(Y-X))*(X**N-1);
(D12)                         0

(C13) EXTEND(ANS,T,3);
                                  2            2
            ((Q - 1) U  - Q + 1) N T
(D13)     - ------------------------
                        2 Q

          2              3          2                  3
   ((2 Q  - 6 Q + 3) U  + (- 2 Q  + 6 Q - 3) U) N T
 - ------------------------------------------------- + ...
                              2
                           6 Q
```

4.3.2. MACSYMA Implementation

The MACSYMA power series package relies on the rational function package to do all of its coefficient arithmetic. Thus the power series can have algebraic function coefficients and coefficients over finite fields with no additional complications to the power series manipulation routines. Consider what information is needed in the internal representation of the power series. We assert that in addition to the exponents and

coefficient pairs, it is also necessary to retain the order of expansion with respect to each of the variables. Thus we are led to the utilize the following list structure.

```
(PS (<var> . <var num>) (<trunc-level>)
    (<exp1> . <coef1>) (<exp2> . <coef2>) ... )
```

This expression represents a univariate power series in the variable <var> expanded to degree <trunc-level>, with coefficient-exponent pairs as shown and ordered such that <expi> less than <expj> if $i < j$. The exponents are represented as a dotted pair of their numerator and denominator. Thus *1/2* would be represented as (1 . 2). The coefficients are represented similarly. The PS at the beginning serves to distinguish the expression from one in the coefficient domain. Multivariate power series can be implemented using the "natural method" by permitting the coefficients to be power series in another variable. To ensure that the representation is canonical an order must be placed on the variables to indicate which may be used as coefficients of which. This ordering is determined by the <var num>. A variable may have as coefficients only those variables whose <var num> is less then their own.

Notice that with this representation a polynomial in several variables need not have a single order of expansion for each variable. For example in the following computation:

```
(C14) TAYLOR(1+X+Y/X,Y,0,1,X,0,1);
                                Y
(D14)                  1 + X + - + . . .
                                X


(C15) TAYLOR(1+2*X+X^2+(1/X+2)*Y,Y,0,1,X,0,2);
                          2    1
(D15)      1 + 2 X + X  + (- + 2 + . . .) Y + . . .
                         X
```

The coefficient of Y is valid to degree zero while the constant term is good to degree two. The expression in D15 would be represented by

```
(PS (y . 2) ((1 . 1))
    ((0 . 1) . (PS (x . 1) ((2 . 1))
                   ((0 . 1) . (1 . 1))
                   ((1 . 1) . (2 . 1))
                   ((2 . 1) . (1 . 1))))
    ((1 . 1) . (PS (x . 1) ((0 . 1))
                   ((-1 . 1) . (2 . 1))
                   ((0 . 1) . (2 . 1)))))
```

The point of expansion, the canonical order of expansion and other bits of information that are accumulated about each variable are kept on a global variable TLIST.

The extendable power series are implemented by a technique which amounts to the classical "funarg" of LISP [14]. All routines which do arithmetic on power series, do their computation in a serial manner. When they are finished they store the information they would need to restart themselves on the power series itself. If the user requests that more terms be generated, by a call to EXTEND, then EXTEND looks at the data in the power series itself and initiates the restart process. Restarting one of the low level routines will generally cause others to also be restarted, all the way up the computational path to expressions that are known exactly.

The internal representation of an extendable power series is merely an extension of the representation for a normal power series, i.e.,

```
(PS (<var> . <var num>)
    (<trunc-level> <extend-routine> <ext-env>)
    (<exp1> . <coef1>) (<exp2> . <coef2>) ... )
```

where <extend-routine> is the particular routine needed to restart this computation. The slot for <ext-env> is usually a list of the particular values <extend-routine> needs to restore the environment. Consider the expansion process for the expression $z + z \sin z$. Recuring to the lowest level, we are faced with the expansion of z to order 5. This would yield

```
EX1 = (PS (z . 1) (∞) ((1 . 1) . (1 . 1)))
```

where ∞ is some internal representation of infinity. This means that the current expression is exact, if it is asked to be extended it just returns itself, it is already as accurate as it can be. On the other hand the computation of $\sin z$ can never be exact, and it would produce an expression like:

```
EX2 = (PS (z . 1) ((5 . 1) compute-sin (5 120))
          . <terms> )
```

In this expression note that the environment contains 120, one of the partial results when computing the factorials, which go in the denominator while generating terms in the expansion. Continuing with this process we would finally get:

```
(PS (z . 1)
    ((5 . 1) compute-plus
             ((PS (z . 1)
                  ((5 . 1) compute-times
                   (EX1 EX2))
                  . <terms1>)
              EX1))
    . <terms2>)
```

Notice that the only portion of this expression which the user is interested in is <terms2>. When trying to extend this expression, say to order *7*, the "extender" would invoke compute-plus who would decide that its first argument needed more terms. This would cause compute-times to be invoked and so on until arriving at an expression like EX1 which cannot be extended, or one like EX2 which depends upon no one else to extend itself.

Because this technique can use up horrendous amounts of storage (it essentially never throws away a result) the scheme outline above is implemented by using very self-referent list structure. Thus if a $\sin z$ is ever needed again, the copy of the expansion which was already generated will be used. Furthermore the environments are really not as simply laid out as shown here, but rather maintains quite a few pointers into the arguments to aid in the computation.

Conclusions

We feel that power series will become one of the dominant techniques in symbolic manipulation of non-rational functions, much as the p-adic techniques have come to dominate polynomial manipualtion. But before this will happen a thorough understanding of how power series are manipulated and how they may be generalized is necessary. Here we hope to have laid some of the ground work, and at least pointed out just how general the power series representation can be made.

Perhaps the most exciting possibility for future research is in the development of simplification relations among special functions. The key point to realize is that if a function is not zero then its power series expansion must have a non-zero term. If we can determine *a priori* what the minimal order of such a term must be, then we can determine all possible relations of a certain form, by throwing in a few parameters and expanding. We will get a series of non-linear equations and the solutions of those equations would be the only possible coefficients. The main problems thus lie in determining what the minimal order must be and determining a power series from a differential or integral equation.

I would like to acknowledge the help Prof. Joel Moses and Barry Trager have given towards my understanding of this problem domain. Jeff Golden and Guy Steele have aided immeasurably in the implementation of the power series system.

Work herein was supported in part by the United States Energy Research and Development Administration contract #E(11-1)-3070.

References

1. *ALTRAN Users Manual*, Bell Telephone Labs, Murray Hill, N.J., 1973

2. Chang, Y.F., "Automatic Solution of Differential Equations," in Lecture Notes in Math. no 430, Springer Verlag, New York, 1975.

3. Euler, B. *Introductio in Analysin Infinitorium*, Lausanne, 1748.

4. _____, *Einleitung in die Analysis des Unendlichen, aus dem Lateinischen ubersetzt von J.A.C.Michelsen*, Berlin, 2 Vols, 1788.

5. Forsyth, A.R., *Theory of Functions of a Complex Variable*, Cambridge Univ. Press, 1918.

6. _____, *Theory of functions of a Complex Variable*, reprint of [5], Dover, New York, 1965.

7. Gentleman, W.M., "Experience with Truncated Power Series," *SIGSAM Bulletin*, vol 8, no 3, August 1974.

8. Gould, "Coefficient Identities for power of Taylor and Dirichlet Series," *American Mathematical Monthy*, vol 81, pp 3-14, 1974.

9. von Holdt, "Rational powers of power series," *American Mathematical Monthy*, vol 72, pp 740-743, 1965.

10. Knuth, D.E., *The Art of Computer Programming*, vol I, Addison-Wesley, New York, 1968.

11. _____, *The Art of Computer Programming*, vol II, Addison-Wesley, New York, 1971.

12. Laurent, J-P., "A Heuristic Program to Compute Limits," *Artificial Intelligence*, vol 4, 69-94, 1973.

13. *MACSYMA Reference Manual*, Mathlab Group, Project MAC, M.I.T., Cambridge, Mass., November 1975.

14. McCarthy, J., et al., *LISP 1.6 Programmer's Manual*, M.I.T. Press, January 1968.

15. Norman, A.C., SCRATCHPAD Power Series Package," *SIGSAM Bulletin*, ACM, New York, vol 9, No. 1, Feb. 1975.

16. Olson, A.M and Ball, W.E., *An Investigation of Some Extensions of a Method for Generating Power Series*, Computer Systems Lab.,Washington University Tech Report No. 12, St. Louis, Mo., April 1969.

17. Phragmin, *Acta Mathematica*, Tome viii, pp 33-42, 1885.

18. Picard, "Memoire sur les fonctions entieres," *Annales de l'Ecole Normale Superieur*, 2me Ser., tome ix, 1880, pp 145-166.

19. Weierstrass, *Mathematische Werke*, mayer und Muller, Berlin, 1895.

20. Zippel, R.E., "A MACSYMA Solution to Problem 8," *SIGSAM Bulletin*, ACM, New York, vol 9, no 1, Feb. 1975.

21. _____, "Multivariate Power Series Expansions," in preparation.

The MACSYMA "Big-Floating-Point" Arithmetic System

Richard J. Fateman
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California at Berkeley
Berkeley, California 94720

Abstract: We describe the design of a set of elementary function floating point evaluation routines for use with an algebraic manipulation system.

Introduction and Motivation

A number of algebraic manipulation systems have avoided floating point arithmetic by using only integers and rational numbers as models of the real number system. Some more advanced systems permit exact computation with algebraic numbers, transcendental functions and interval arithmetic, but may still assume numerical work is best done in FORTRAN, or some similar language. Some systems have adopted a compromise in the direction of numerical work: floating point approximations to real numbers are acceptable in some contexts. Usually this means that one can use the single or double precision floating point representation available in the host computer. Here we describe the approach developed for use in MACSYMA [3]. It is somewhat distinct in philosophy from the SAC-I multiple-precision work of Pinkert [4] and the many FORTRAN-extension type systems represented, for example, by [5] in that non-rational functions are accurately computed to full precision.

There are several obvious uses for a floating point number system within an algebraic manipulation system.

(1) When exact rational answers are not necessary, the floating-point approximation is typically easier to read, write, compute with, and comprehend. For example, 3900231685776981/2251799813685248 is approximately 1.7320508.

(2) Systems allowing non-rational functions such as sine, logarithm, etc., are unable to evaluate arbitrary constant expressions to an exact numeric (real) value anyway, and so floating-point approximations are frequently as appropriate or more appropriate than rational approximations.

(3) Typical scientific calculations may begin with floating-point approximations as data; conversion to exact rational arithmetic may be an unnecessary and counterproductive transformation.

There are several problems associated with the use of the usual floating-point representation within algebraic manipulation systems.

(1) Systems with arbitrary precision integer arithmetic can produce integers not representable as floating-point numbers because of the limit on exponent size.

(2) The number of digits of precision available may be insufficient for the purposes at hand.

(3) The hardware operations and/or the library of numerical algorithms available may be unsatisfactory.

The Approach in MACSYMA

MACSYMA [2] has incorporated the ordinary single-precision floating-point operations, and a set of evaluation routines into its regular PDP-10 system in order to facilitate casual "evaluation" of a real-valued function. Additionally, large scale, more cost-effective computations can be done by transferring purely numerical work to an efficient floating-point processor (eg. IBM 360/91). However, a new data type, the big floating-point number (or "bigfloat") was added to MACSYMA in the Spring of 1974 to take advantage of situations in which floating-point representation is useful, and yet usual floating-point precision is not acceptable. Of course, higher precision routines will generally have (much) higher costs, but that cost may be insignificant compared to the value of having computations done (much) more precisely.

A MACSYMA bigfloat is a LISP S-expression of the form ((BIGFLOAT SIMP p)m e) which represents the number $m*2^{e-p}$. The precision is p (bits), the exponent is e, and m is the mantissa. The mantissa has an implied binary point to the left of its left-most significant bit. Both e and m are represented as arbitrary precision integers, and may be positive or negative. The precision p must be a positive integer. A bigfloat is ordinarily normalized, which means that m has p significant bits, or is 0 (which by convention has m = e = 0). Base 2 is used primarily to facilitate shifting operations; however, several operations can be done in base-10 arithmetic to facilitate the computation of output formats. Unnormalized arithmetic is not supported, but could be added.

A bigfloat may be created in MACSYMA by taking any MACSYMA rational number, integer, single precision floating point number, or bigfloat (perhaps of the wrong precision), say n, and computing BFLOAT(n). In fact, n need not be a number, but can be essentially any expression which evaluates to a constant by a recursive application of

---

+A talk based on this paper was presented at the SIAM-SIGNUM 1975 Fall Meeting, San Francisco, December 1975.

bigfloat evaluations of its constituents. Thus BFLOAT(SIN(1)/COS(1)) is computed as BFLOAT(SIN(1))/BFLOAT(COS(1)). A bigfloat may be typed in directly as in the syntax of FORTRAN, the distinguishing mark being a "B" instead of an "E" to introduce the exponent. Thus, 12.01B-2 is an acceptable bigfloat input token. The output format is similar.

The user declares the floating-point precision to be used by the command FPPREC(Q); where Q is the number of decimal digits which will be presented in output (or accepted as input). This is implemented by internally carrying a binary precision $p > \log_2(10^Q)$ bits. Currently, $p = 2 + \lceil \log_2(10^Q) \rceil$ in order to encode slightly more than Q decimal digits on input, and provide simpler rounding before output.

By altering the global precision, values of different precisions may coexist, and a particular variable may assume values of different precisions during a computation. Each operation is done as though each of its operands is accurate to the gloval precision (even if its precision is less), and each value stored is of the global precision at that time. This seems more reasonable than, for example, declaring the precision of a name (eg. FORTRAN's "DOUBLE PRECISION X") since this fails to specify how to compute the right hand side of X ← B/(C+D). Even if C and D are single precision, the sum perhaps should be more accurately computed. Furthermore, there is no way to tell from the history of a number, how accurately subsequent computations with it should be done [2].

Arithmetic routines are provided for the ordinary arithmetic operations, exponentiation, the exponential function, the natural logarithm, sine, cosine and tan. The hyperbolic, inverse-hyperbolic, and inverse-circular functions can be computed by composition of these functions, but care must be taken to avoid loss of accuracy due to cancellation.

It is our intention that the answer supplied by any of the provided algorithms is correctly rounded to the nearest unit in the last place, or is at most one bit off. This is usually done in the case of the transcendental functions by computing the answer to some calculated number of bits higher precision, and then rounding. In general, it is necessary to increase the precision of the internal calculations not only on the basis of the required output precision, but also on the argument itself. Thus in order to compute sine of certain "bad" angles, considerable time and effort is expended.

Among the programs supplied to provide these features are functions which generate π and e to arbitrary precision. In fact, the programs compute and save these values when the programs are first loaded, although recomputation to higher precision may be required when the user invokes procedures at particular arguments. For example, to get 10 digits of sin(x) where x agrees with π to 10 digits, requires at least 20 digits of π.

While some effort has been expended to choose the best machine-independent algorithms for each of these operations, more efficient ones may be discovered. In fact, it is our intent to describe all these algorithms in sufficient detail in an entirely machine-independent fashion, so that the substitution of faster, cleaner, or shorter algorithms can be done whenever an improvement to the given algorithm is discovered. For example, the recent work of Brent [1] should be considered for transcendental function calculation of precision higher than some implementation-dependent cut-off point. The only unusual prerequisite for the host system is a data type of "arbitrary precision integer." These programs are in Appendix 1.

These programs have been incorporated into MACSYMA mostly through modification of the "general" simplifier. Bigfloats have been made "contagious" through *, /, +, **, and SIN, COS, etc. That is, if a bigfloat appears in a sum, for example, all other numbers (integers, rational, floating-point, or bigfloats) will be converted to bigfloats and added together. Setting a global switch, BFLOAT, to TRUE forces all rational numbers to become bigfloats, and non-rational functions of numeric arguments (eg. integer, rational) will be calculated as bigfloats. Conversion from a bigfloat to a rational approximation may be done by using the RAT command. Conversion from a bigfloat to a single-precision floating-point number may be done by using the FLOAT command. Of course, not every bigfloat can be so converted, since the exponent range for bigfloats is larger.

Since the floating point precision may be changed at any time, bigfloats of differing precision may coexist. If two such numbers are operated on in some way to combine them, they are each converted to the current precision by concatenation of 0's to the binary mantissa, or by dropping off lower-order digits (with rounding) before the operation is carried out.

Where Next?

Bigfloats are a logical building block for several features we hope to see incorporated in MACSYMA, namely interval arithmetic and bigfloat complex numbers. We expect that these implementations will again emphasize the need of obtaining a best possible answer where one exists, rather than a "reasonably good" answer. We also expect that guaranteed error bounds plus the option of always being able to carry more precision will lead to greater confidence in computed results.

Speed improvements may be attainable through modifications of algorithms for division, the exponential, logarithm, and trigonometric functions. The computational complexity arguments of Brent [1] along these lines seems especially promising. By computing answers using (sometimes much) higher internal precision in order to insure accuracy, we are serving a different constituency than that of [5] or earlier systems for triple-precision (etc.) routines. The use of FORTRAN-language extensions for high precision calculations has a long history, but typically high-precision and high accuracy are not linked as they are here and in [1]. We hope this paper serves to stimulate activities along these lines and also provokes further pursuit of scientific inquiries where very accurate floating point computations have been found necessary.

Acknowledgments

I wish to thank Prof. W. Kahan for many valuable suggestions.

This work was supported in part by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract N00014-70-A-0362-0001, and in part by the Energy Research and Development Administration through the Lawrence Livermore Laboratory.

Appendix 1

The descriptions of the programs below are detailed only in those instances where the method might be improved in terms of accuracy, speed, or both. Programs which are particularly machine-dependent are only briefly sketched. The descriptions are paraphrases of the existing LISP programs. A semi-colon is used to indicate the beginning of a comment, which is terminated by the end-of-line. Statements are separated by commas and may also be grouped into "begin-end" blocks by parentheses. Steps for which the exact programming implementation is obscure are described within brackets: { }. The MACLISP programs for these functions are available from the author, and may be run directly with available software on a PDP-10 TENEX operating system, a standard DEC 10/50 operating system, or on M.I.T.'s ITS system. They may also be run on standard Multics systems.

Arithmetic operations such as "+" or "*" are used to denote operations on either integers of arbitrary precision or bigfloats. The context distinguishes the two cases. Where the types of numbers are mixed, the integers will always be converted to bigfloats. Throughout the descriptions the floating point arithmetic is performed to a precision specified by a global variable *fpprec*. The precision of the operation is not affected by the precision of the inputs.

intofp(x): given an integer x returns its bigfloat representation.

fixfloat(x): given a single-precision floating-point number x returns a lisp pair (*numerator* . *denominator*) which is the exact rational representation for x.

floattofp(x): given a single-precision floating-point number x computes *fixfloat(x)*, then returns *fpquotient*(*intofp*(*numerator*),*intofp*(*denominator*)).

bfloat2rat(x): given a bigfloat x returns the exact rational representation of x as (*numerator* . *denominator*).

fpformat1(x): given a positive bigfloat x returns a character string of appropriate length representing the closest decimal bigfloat representation.

fpscan(lft,rt,exp): returns a bigfloat representation. It is used by the lexical (input) scanner and computes the bigfloat corresponding to "*lft*"."*rt*"B"*exp*" on input. For example, 12.01B-3 would set *lft* = '(/1 /2), *rt* = '(/0 /1), *exp* = '(-/3). Each of these components is a list of character objects using the MACLISP printing conventions. The answer is correct to all places if *length*(*lft*) + *length*(*rt*) $\leq$ Q, the floating-point precision. The answer is computed by first calculating the appropriate number of extra bits required to compute 10^{exp} correctly. In more detail, the algorithm is given below, where *readlist* converts a list of digits in base 10 into a number.

```
fpscan(lft,rt,exp) :=
{set precision to current precision
   + log2(readlist(exp))
   + fix(1 + 3.322*length(lft))},
ans ← bfloat(10**readlist(exp)*
      (readlist(lft)+readlist(rt)/length(rt))),
{set precision back},
return(ans).
```

Normalization

fpshift(l,n): takes an arbitrary precision integer l, and returns the result of (left) shifting l n bits. If n is negative, a right shift is done. If the global flag **decfp* is TRUE, then l is considered to be a decimal number, and n decimal digits are shifted.

fpround(l): takes an arbitrary precision integer l of at least *fpprec+1* bits, and returns l rounded to a mantissa of *fpprec* bits. A global variable *mm* is set to the number of bits shifted. Rounding is achieved by adding a bit one place to the right of the last bit to be retained, and then chopping using *fpshift*. In the case where the digits chopped off are all zeros, the last bit retained is forced to be zero. That is, in case the number is exactly mid-way between two representable numbers, the "even" one is chosen. This is done through the use of a "sticky" bit as suggested by W. Kahan [2]. This operation can also be done in a decimal base.

```
fpround(l) :=
mm ← howlong(l)-fpprec, ;"howlong" returns the
                        ;number of significant
                        ;bits in |l|
adjust ← fpshift(1,mm-1), if l<0 then
   adjust ← -adjust, ;adjust is the rounding
                     ;bit
l ← l+adjust, mm ← howlong(l)-fpprec,
stickybit ← highpart(l,mm), ;stickybit is the
                            ;rightmost mm bits
if stickybit = 0 then return
   (fpshift(fpshift(l,1-mm))-1 ;round to even
else return (fpshift(l,-mm)).
```

Rational Operations

fpplus(x,y): adds x and y to get a rounded floating-point answer correct in all digits. This is done by carrying a guard digit and a "sticky bit," the role of which is to insure that if the answer is half-way between two representable numbers, the answer is rounded to the even number. This is illustrated in *fpround*, above. The number of "cancelled" bits is made available through a global variable.

fpdifference(x,y): = *fpplus(x,fpminus(y))*

fpminus(x): changes the sign of x

fptimes(x,y): multiples the mantissas and adds the exponents, then rounds. Currently the full (*2*fpprec* precision) product is developed, and then rounded to precision *fpprec*.

fpquotient(x,y): checks for x or y zero, otherwise leftshifts the mantissa of x by *fpprec+3* bits, divides by the mantissa of y, computes the appropriate exponent, and uses *fpround* to normalize and round.

fpexpt(x,n): computes a power. There are two cases for raising to a power: n an integer and n a bigfloat. Computing an integer power is done by a standard binary-expansion-of-the-exponent chain of multiplications and squarings. The case n is a bigfloat is discussed under the Exponential and Logarithm section. The former case is trivial, and is not worth detailing.

fpabs(x): converts the sign of the mantissa to positive.

fpintpart(x): returns an integer which is the (signed) integer to the left of the decimal (or binary) point in the representation.

fpgreaterp(x,y): returns true if *fpdifference(x,y)* is greater than 0.

fplessp(x,y): = *fpgreaterp(y,x)*.

Circular Functions

fppi( ): returns the value of π to current precision *fpprec*. *fppi( )* examines the precision of the last saved value of π, and if it is sufficient, returns the rounded answer. It also resets the saved value to the precision supplied. If the precision saved is not sufficient, the mantissa is computed by the following algorithm to *fpprec+3* bits, then rounded:

```
comppi1(n) := ;computes π to n bits (as an
              ;appropriately scaled integer)
a ← 2**n, c ← 3*a+(b ← a//8), ;// is integer
                              ;quotient
for i from 4 by 2 until b=0 do
(b ← b*(i-1)**2//(4*i*(i+1)), c ← c+b),
return (c). ;c/2**n is π.
```

fpsin(x): returns *sin(x)* computed by means of a Taylor series approximation of x translated into the interval $[0,\pi/2]$. The argument x will be increased in precision if necessary to allow it to be placed into the standard interval with no error. (This requires π to be computed to high precision if x is of large magnitude). The Taylor series is evaluated in a standard fashion, carrying 3 bits additional precision. The answer is returned rounded down to *fpprec* bits. There is no error for "argument out of range," since this is not possible.

```
fpsin1(x) := ;Taylor series for sine used when
             ;0 ≤ x ≤ π/2
local(ans,x2,term,oans),
ans ← x, x2 ← -x*x, term ← x, oans ← false,
for n from 3 step 2 until ans = oans do
   (term ← term *x2/(n*(n-1)),
   oans ← ans,
   ans ← ans + term),
return(ans).
```

```
fpsin(x) :=
local(oldprec,sign,lostbits,piby2,n,result)
   sign ← if x>0 then 1 else -1,
   x ← fpabs(x),
   if x=0 then return(0),
   oldprec ← fpprec ;save old precision
   fpprec ← max(fpprec,fpprec + exponent of (x))
      + 3,
   lostbits ← 0,
   repeat fpprec ← fpprec + lostbits,
          piby2 ← fppi( )/2,
          n ← fpintpart(x/piby2),
          y ← x-n*piby2,
          lostbits ← {number of bits cancelled
             in last subtraction},
             x-(n+1)*piby2,
          lostbits ← max(lostbits,{number of
             bits cancelled in last subtrac-
             tion}),
   until fpprec - lostbits ≥ oldprec,
      ;insure enough bits
   r ← mod(n,4), ;compute right quadrant
   fpprec ← oldprec+3, ;reduce precision back
   result ← sign(if r=0 then fpsin1(x) else
                 if r=1 then fpcos1(x) else
                    ;fpcos1 is like fpsin1 but
                    ;computes cosine
                 if r=2 then -fpsin1(x) else
                 if r=3 then -fpcos1(x))
   fpprec ← oldprec, ;restore precision
   return(fpround(result)).
```

fpcos(x): returns *cos(x)* computed by a slight modification of *fpsin*, and has the same structure (in fact, it is the same program, with a "flag" to test whether sin or cos is being asked for).

```
fpatan(x) :=
{precision increased to fpprec+8 internally}
if fpabs(x) > 1 then return (pi/2-fpatan(1/x)),
if fpabs(x) > 1/2 then
   ; x/(1+x^2)*{1+(2/3)*(x^2/(1+x^2))+···}
   (tmp←x*(1+x*x), x2 ← x*tmp, term ← 1, ans ← 1,
      oans ← false,
   for n from 0 step 1 until ans = oans do
   (term ← term*x2*(2*n+2)/(2*n+3), oans ← ans,
      ans ← ans + term),
   return(ans*tmp))
else ;x-x^3/3+x^5/5-...
   (ans ← x, x2 ← -x*x, term ← x, oans ← false,
   for n from 3 step 2 until ans = oans do
   (term ← term*x2, oans ← ans, ans ← ans + term/n),
   return(ans))
```

Exponential and Logarithm

fpe( ): returns the value of e to current precision. This is done by examining the previously saved value (as in *fppi* above), or if that is insufficiently precise, computing *fpexp(1)*, where *fpexp* is described below:

```
fpexp(x) :=
local(r,result,oldprec)
{precision increased to p+8 internally}
if x<0 then return (1/fpexp(-x)),
r←fpintpart(x),
if r<2 then return (fpexpl(x)),
oldprec←fpprec,
fpprec←fpprec+⌈log2(r)⌉ ;compute e to higher
                        ;accuracy,
result←fpexpt(fpe( ),r),
fpprec←oldprec, ;restore precision
result←result*fpexpl(x-r), ;e^x = e^r e^(x-r)
return(result).

fpexpl(x) := ;Taylor series is used only for
             ;arguments in [0,1]
term←1, ans←1, oans←false,
for n from 1 step 1 until ans=oans do
(term←x*term/n, oans←ans, ans←ans+term),
return(ans).
```

fplog(x): returns the natural log of x by first reducing x to the range $[1/e,e]$ by divisions or multiplications (as above in *fpexp*) and then uses

```
fplogl(x) := x←x-1, term←x/(x+2),
             x←term*term, oans←false,
   for n from 0 step 1 until ans=oans do
   (ans←ans+2*term/(2*n+1), term←term*x),
   return(ans).
```

fppower(x,y): computes x^y where x and y are *fpprec*-precision bigfloats. If x is negative, a root of (-1) is divided out. The answer is correct to precision *fpprec* even if this requires higher-than-*fpprec* precision arithmetic to compute *fpexp(y*fplog(x))*. It will use lower-than-*fpprec* precision for this computation when possible. Essentially, *fpexp(y*fplog(x))* is computed where *y*fplog(x)* is computed to precision $log_2(y) + log_2(log_e(x))$.

Appendix 2

Examples

user commands in italics

```
(C1) FPPREC(24)$   {set precision to 24 places}
(C2) SIN(X)↑3+COS(X↑2)↑4;
(D2) COS⁴(X²)+SIN³(X)
(C3) D2,X=1.23456789012345678901B12;
       {evaluate expression D2 to 24 places}
(D3) -5.22379381617523486512642B-2
(C4) %PI,BFLOAT;   {π to 24 places}
(D4)  3.14159265358979323846264B0
(C5) SIN(D4);   {This requires π to be computed to
                 48+ places!}
(D5) -3.80697792403865321819264B-25
(C6) COS(D4);
(D6) -1.0B0
(C7) ATAN(D5/D6);
(D7)  3.80697792403865321819264B-25
(C8) SIN(1.0B0);
(D8)  8.41470984807896506652503B-1
(C9) COS(1.0B0);
(D9)  5.40302305868139717400937B-1
(C10) ATAN(D8/D9);
(D10) 1.0B0
(C11)
```

References

[1] Brent, R.P., "Fast multiple-precision evaluation of elementary functions," *J. ACM 23*, 2 (April 1976), 242-251. Also "A Fortran multiple-precision arithmetic package," submitted to a technical journal.

[2] Kahan, W., "Implementation of Algorithms," Technical Report 20, Computer Science Department, University of California, Berkeley, 1973. Also "A survey of error analysis," *Proc. IFIP 71*, North Holland Publ. Co., 1971, 1214-1239.

[3] Mathlab Group, "The MACSYMA Manual," Laboratory for Computer Science, Massachusetts Institute of Technology, 1974, 1976.

[4] Pinkert, J.R., "SAC-1 variable precision floating point arithmetic," *Proc. ACM 75*, 274-276.

[5] Wyatt, W.T. Jr., D.W. Lozier, D.J. Orsen, "A portable extended precision arithmetic package and library with Fortran precompiler," presented at Mathematical Software II, Purdue University, May 1974.

Interval Arithmetic Applied to Polynomial Remainder Sequences

James R. Pinkert
The University of Tennessee, Computer Science Department
Knoxville, Tennessee 37916

Abstract

Polynomial remainder sequences are the basis of many important algorithms in symbolic and algebraic manipulation. In a number of these algorithms, the actual coefficients of the sequence are not required; rather, the method uses the signs of the coefficients. Present techniques, however, compute the exact coefficients (or a mixed radix representation of them), and then obtain the signs. This paper discusses a new approach in which interval arithmetic is used to obtain the signs of the coefficients without computing their exact values. Comparisons of this method with analogous standard techniques show empirical computing time reductions of two orders of magnitude for even relatively small cases.

1. Introduction

Two occurrences at Mathematical Software II [PMS74] were especially important to this paper. One was Oliver Aberth's presentation of some very interesting techniques in interval arithmetic. The other was a discussion with George E. Collins regarding potential time reductions in algorithms of symbolic and algebraic manipulation if approximate or interval arithmetic could replace some of the exact arithmetic.

One of the author's main interests is the computation of polynomial roots using algebraic algorithms. A major tool in such work is the integer polynomial remainder sequence [PIJ73].

It is important to note that in this application the exact numerical values of coefficients in a given sequence are not required. Rather, one is concerned with the signs of the coefficients.

Two algorithms for computing these signs have received considerable attention [PIJ73]. In one, pseudo-division [COG71] is used to compute a reduced integer polynomial remainder sequence; the signs are then obtained from the integer coefficients. (A greatest common divisor algorithm can be added to obtain a primitive integer polynomial remainder sequence.) In the other algorithm, modular methods are used to obtain mixed radix representations [KND69] of the coefficients. The signs can then be obtained from these representations without actually computing the integer values. Since signs, and not numeric values, are required, the considerations brought up in the first paragraph of this section seemed particularly relevant to these computations.

A SAC-1 [PIJ73] interval arithmetic package was implemented, employing some concepts from Aberth's presentation. The standard SAC-1 algorithm for computing the pseudo-remainder of two polynomials [COG71] was then modified to use this interval arithmetic.

Empirical comparisons of the standard and modified versions have been very encouraging. Time reductions of two orders of magnitude have been realized on relatively small problems, and extrapolations seem to indicate even greater reductions for larger cases. For example, one list of signs which could not be computed in thirty minutes using the standard pseudo-division algorithm was obtained in one second using the interval arithmetic version.

Section 2 of this paper discusses the modified polynomial remainder sequence algorithm. Sample empirical computing times are shown in Section 3. Section 4 summarizes the results.

2. Pseudo-Remainders and Sign Lists

This section describes an algorithm which uses pseudo-remainders to compute one standard type of coefficient sign list. The discussion is abbreviated, since the methods have appeared previously [COG71, PIJ73].

A SAC-1 interval polynomial has exactly the same internal representation as an integer polynomial [COG71], except that the numerical coefficients are intervals rather than integers. Hence a pseudo-remainder algorithm can be applied to either type of polynomial simply by loading the corresponding arithmetic routines.

The following algorithm applies pseudo-remainders to compute the list of degrees and leading coefficient signs in the reduced polynomial remainder sequence for polynomials P and Q.

List of Signs and Degrees, lsad:

Step 1: L←(sign{ldcf{P}},deg{P},sign{ldcf{Q}},deg{Q});

```
Step 2: While (R ≠ 0) & (d ≠ 0) & (0 ∉ r) do;
            R ← psrem{P, Q};
            if R ≠ 0 then do;
                 r ← ldcf{R};
                 if 0 ∉ r then do;
                      s ← sign{r};
                      d ← deg{R};
                      L ← L||(s, d);
                      end;
                   else L ← ();
                 end;
            P ← Q;
            Q ← R;
            end;
Step 3: Return L.
```

Note that there are two return conditions. A successful return is executed when the pseudo-remainder is zero or is of degree zero. A failure return is executed when the sign of the leading coefficient cannot be determined. By definition, if $R \neq 0$ then $r = ldcf\{R\} \neq 0$. Hence if $0 \in r$ then the failure condition has occurred. The cause, of course, is that the intervals have not been carried to a sufficient number of β-digits.

The latter algorithm can be used with integer arithmetic simply be removing the test for $0 \in r$:

```
Step 2: While (R ≠ 0) & (d ≠ 0) do;
            R ← psrem{P, Q};
            if R≠0 then L←L||(sign{ldcf{R}},deg{R});
            P ← Q;
            Q ← R;
            end;
```

3. Comparison of Empirical Computing Times

The algorithm for computing the list of signs and degrees, lsad, was applied to randomly generated polynomials, first using integer methods and then using interval methods. The empirical results are shown in this section.

Random polynomial P, of specified degree and infinity norm, was obtained. Since lsad{P, P´} is used often [PIJ73], this case was run first. Then, for $1 \leq j \leq deg\{P\}$, random polynomial Q of degree j was generated, and lsad{P, Q} was computed.

This process was repeated for another polynomial P of the same degree and norm. The exact number of repetitions for a given entry depended on the time required for each repetition.

The average results are shown in Tables 1, 2, 3, and 4. In each comparison, the first entry of the pair is the time (in milliseconds) required for the integer method. The second entry is the time (in milliseconds) required for the interval method, using the minimum possible number of β-digits. A question mark indicates that the set of runs could not be completed in a reasonable amount of time.

The time comparisons were made with reduced rather than primitive integer techniques because of the direct analogy in algorithms. Modular methods were not included because, unlike many other instances, they have not resulted in any consistent improvements over integer methods [PIJ73].

Note that the times shown are with intervals carried to the minimum number of β-digits. How does one determine this minimum? As yet, the author has found no way of deriving any theoretical answers to the question. For the sample runs discussed, the minimums were determined experimentally. Given this situation, the times shown for the interval method should included overhead resulting from using too few or too many β-digits.

A small percentage of the test runs required five β-digits; all others required four or fewer. This result is very interesting, since coefficients of a hundred β-digits were not uncommon in the integer computations. It suggests an algorithm which simply starts with one β-digit, and increments by one until the list is computed. This approach was tried, and it gave an average increase of two to three times the illustrated results.

The observed minimum number of β-digits seemed to depend mainly on the degree of Q (or the degree of P´). Hence the author started at this value and doubled the number of β-digits used each time a failure occurred. This approach yielded almost exactly the same set of times as shown in the tables.

$L(|P|_\infty), L(|Q|_\infty) = 5$

degree of P	P´	degree of Q				
		5	4	3	2	1
6	82086	81674	26150	7062	2038	1020
	822	931	607	424	332	203
5	16407		14743	4559	1506	707
	590		416	416	241	138
4	3195			3122	1040	441
	324			291	183	91
3	636				519	274
	133				103	83
2	141					150
	50					33

Table 1

$L(|P|_\infty), L(|Q|_\infty) = 10$

degree of P	P´	degree of Q				
		5	4	3	2	1
6	353547	335427	101117	27564	7572	3470
	949	866	783	707	325	241
5	61244		56543	17614	4884	2147
	541		475	292	217	133
4	10766			10725	3437	1423
	400			250	242	117
3	1939				1889	774
	100				125	67
2	341					349
	50					50

Table 2

$L(|P|_\infty), L(|Q|_\infty) = 15$

degree of P	P	degree of Q				
		5	4	3	2	1
6	?	?	?	?	?	?
	965	858	690	607	316	250
5	131714		142139	38763	11140	4768
	516		566	433	292	150
4	22764			22423	6614	2515
	350			317	191	100
3	3969				3944	1423
	125				133	83
2	757					683
	50					42

Table 3

$L(|P|_\infty), L(|Q|_\infty) = 20$

degree of P	P	degree of Q				
		5	4	3	2	1
6	?	?	?	?	?	?
	940	732	649	541	366	225
5	?		?	?	?	?
	541		566	391	250	133
4	41958			42216	11781	4568
	275			300	208	108
3	6898				6623	2438
	125				133	84
2	1157					1206
	59					33

Table 4

5. Summary

Interval methods appear to have great potential in the computation of polynomial remainder sequence coefficient signs.

Obviously, more research needs to be done in determining how many β-digits to carry for the intervals. However, significant improvements seem to be possible even with simplistic approaches.

It is hoped that these results, in addition to being useful per se, might suggest further applications of interval methods in symbolic and algebraic manipulation.

References

[COG68] G. E. Collins and J. R. Pinkert, The Revised SAC-1 Integer Arithmetic System, University of Wisconsin Computing Center Technical Report No. 9, Nov., 1968.

[COG71] G. E. Collins, The SAC-1 Polynomial System, University of Wisconsin Computer Sciences Department Technical Report No. 115, March, 1971.

[KND69] D. E. Knuth, The Art of Computer Programming, Vol. II: Seminumerical Algorithms, Addison-Wesley Publishing Co., Reading, Mass., 1969.

[PIJ73] J. R. Pinkert, Algebraic Algorithms for Computing the Complex Zeros of Gaussian Polynomials, University of Wisconsin Computer Sciences Department Ph.D. Thesis, June, 1973.

[PMS74] Proceedings of Mathematical Software II, Purdue University, May, 1974.

Algebraic Factoring and Rational Function Integration

by

Barry M. Trager

Laboratory for Computer Science, MIT
(formerly Project MAC)

Abstact

This paper presents a new, simple, and efficient algorithm for factoring polynomials in several variables over an algebraic number field. The algorithm is then used iteratively to construct the splitting field of a polynomial over the integers. Finally the factorization and splitting field algorithms are applied to the problem of determining the transcendental part of the integral of a rational function. In particular, a constructive procedure is given for finding the least degree extension field in which the integral can be expressed.

1. Introduction

For many applications in symbolic mathematics it is necessary to explicitly describe all the roots of a polynomial. One approach is to compute the roots numerically to some predetermined accuracy. This is the approach taken in numerical analysis packages but is generally avoided in symbolic manipulations as the basic routines such as greatest common divisor and factoring would then be useless. Another attempt is to try to express the roots in terms of radicals, but this often cannot be done, and even when it can it leads to great problems when simplifications are required. The approach taken here is to describe an extension field of the rationals by the minimal polynomial for some primitive element and then to express all the roots of the polynomials in terms of that element. If an irreducible polynomial over k is normal then all of its roots are rationally expressible in terms of any one of them. Thus the degree of its splitting field is the same as the degree of the polynomial. For other polynomials, however, the degree of its splitting field may be as high as $n!$. This degree growth tends to make many "computable" problems practical impossibilities. It then becomes very important to operate in as low degree an extension field as possible.

2. Norms and Algebraic Factoring

We assume the existence of some base field of characteristic zero (e.g. the rationals) over which we have constructive procedures for factoring polynomials. We also assume the capability is present to perform basic polynomial operations (e.g. division with remainder) in both the base field and some extension field of finite degree. Our approach is to map a polynomial over an extension field to one of higher degree in the base field such that there is an exact correspondence between their factorizations in their respective fields. We use this correspondence to reconstruct the factorization of the original polynomial over the extension field.

2.1. Definitions

A number α which is algebraic over k satisfies an irreducible polynomial with coefficients in k. $k(\alpha)$ is the field obtained by adjoining α to k. Let $f_\alpha(x)$ be the unique monic, irreducible equation of degree n which α satisfies. The conjugates of α over k are the remaining distinct roots of f_α, α_2, α_3, ..., α_n. If β is any element of $k(\alpha)$ then β can be represented uniquely as a polynomial in α, of degree less than n, with coefficients from k, $P_\beta(\alpha)$. (See [5 pp. 91-94]) The conjugates of β considered as an element of $k(\alpha)$ are $P(\alpha_2)$, $P(\alpha_3)$, ..., $P(\alpha_n)$. If $\beta = P(\alpha)$ we will let β_i represent $P(\alpha_i)$.

A very useful mapping from $k(\alpha)$ to k is the Norm. Norm(β) is the product of all the conjugates of β relative to $k(\alpha)$ over k. If we want to emphasize the fields involved we may also write $\text{Norm}_{[k(\alpha)/k]}(\beta)$. Since the norm is symmetric in the α_i, by the fundamental theorem on symmetric functions it can be expressed in terms of the coefficients of P_α and thus lies in k.

We can extend the definition of norm to polynomials in several variables with coefficients in $k(\alpha)$. Any such function can be expressed as $G(x_1,x_2,\ldots,x_k,\alpha)$ where G is a polynomial in several variables with coefficients in k. Then the Norm(G) is the product of $G(x_1,x_2,\ldots,x_k,\alpha_i)$ over the n conjugates of α.

The definition given above for the Norm of G coincides with the definition of Resultant($P_\alpha(y)$,$G(x_1,\ldots,x_k,y)$) as presented in [1 p. 86]. Collins [1] presents a modular resultant algorithm which can be employed to effectively compute norms. Even if the original polynomial is sparse, its norm is likely to be dense; thus a modular algorithm is probably the optimal choice. Note that norm is a multiplicative function by definition, i.e. Norm(A*B)=Norm(A)*Norm(B). Thus we can extend the norm to rational functions by defining Norm(A/B)=Norm(A)/Norm(B).

We are now ready to prove an extremely important property of norms. (See [14 pp. 19-20])

Theorem 2.1: *If $f(x,\alpha)$ is an irreducible polynomial over $k(\alpha)$ then the Norm(f) is a power of an irreducible polynomial over k.*

Proof: Assume Norm(f)=C(x)*D(x) where gcd(C,D)=1. Let $f_1 = f(x,\alpha_1)$, then Norm(f) = Product(f_i). The polynomial $f=f_1$ divides Norm(f) and since f is irreducible, either f|C or f|D. Assume for concreteness that f|C, i.e. $C=f_1*g_1$. The fields $k(\alpha_1)$ and $k(\alpha_j)$ are canonically isomorphic under a mapping ϕ_j which sends α_1 to α_j and is the identity on k. ϕ_j can be extended to the ring of polynomials in x over those fields and still remain an isomorphism. Since C(x) has all its coefficients in k it is invariant under ϕ_j, but f_1 and g_1 are mapped to f_j and g_j respectively. Thus the equation $C=f_1*g_1$ becomes $C=f_j*g_j$ under ϕ_j. Therefore $f_j|C$ for $1\le j\le n$, but gcd(C,D)=1 implies that $\gcd(f_j,D)=1$ for all j, and in turn gcd(product(f_j),D)=1. But we assumed that D|Norm(f)=product(f_j), so we have shown that D=1. Thus the Norm(f) cannot be split into two relatively prime factors and can only be some power of an irreducible polynomial. ∎

A simple application of the above theorem is for finding minimal polynomials for elements of $k(\alpha)$. If β is an element of $k(\alpha)$ then, as above, $\beta=Q(\alpha)$. Thus $x-\beta$ divides the Norm(x-Q(α))=B(x) and therefore $B(\beta)=0$. The problem remaining is to determine which irreducible factor of B(x) is actually the minimal polynomial of β. The polynomial $x-\beta$ is linear and thus obviously irreducible. By the above theorem we see that B(x) can only be a power of the minimal polynomial for β, $B(x)=P_\beta(x)^k$. $P_\beta(x)$ can be found directly by calculating the gcd(B(x),B'(x)) where B'(x) is the derivative of B(x).

We now turn to the problem of factoring polynomials with coefficients in $k(\alpha)$ assuming we have this capability over k. The ability to perform basic arithmetic in $k(\alpha)$ allows one to compute gcd's of polynomials over $k(\alpha)$ by the Euclidean Algorithm. Thus we can perform a square free decomposition (see [15]) on any polynomial and reduce the factoring problem to square free polynomials.

Our approach to the factorization of $f(x,\alpha)$ is first to make a linear substitution for x so that the norm(f) is square free. We then factor the norm(f) over k. $\text{Norm}(f)=G_1(x)\,G_2(x)\,\ldots\,G_r(x)$ with each G_i distinct and irreducible over k. We claim that $g_i(x,\alpha)=\gcd(f,G_i)$ is irreducible over $k(\alpha)$ for all i and that f=product(g_i).

Theorem 2.2: *Let $f(x,\alpha)$ be a polynomial over $k(\alpha)$ such that the Norm(f) is square free. Let $\Pi(G_i(x))$ be a complete factorization of the Norm(f) over k. Then $\Pi(\gcd(f(x,\alpha),G_i(x)))$ is a complete factorization of f over $k(\alpha)$.*

Proof: Let $g_i=\gcd(f(x,\alpha),G_i(x))$, then we must show that each g_i is irreducible and that all the irreducible factors of f are among the g_i. Let v(x) be an irreducible factor of f over $k(\alpha)$. By the previous theorem Norm(v) must be a power of an irreducible polynomial over k, but v|f implies Norm(v)|Norm(f) and the Norm(f) is square free. Therefore the Norm(v) is irreducible and must be one of the G_i. Since the Norm(f) is equal to the product of the norms of each of the irreducible factors of f, each G_j must be the norm of some irreducible factor of f. Assume both $v_1(x)$ and $v_2(x)$ divide $\gcd(f,G_i)$ where v_1 and v_2 are distinct irreducible factors of f. $v_1|G_i$ implies $\text{norm}(v_1)|\text{norm}(G_i)$, but $G_i(x)$ is a polynomial over k and its norm is $G_i(x)^n$. The $\text{norm}(v_1)$ is irreducible over k and divides a power of the irreducible polynomial $G_i(x)$, thus the $\text{norm}(v_1)=G_i(x)$. Similarly the $\text{norm}(v_2)=G_i(x)$. But $(v_1*v_2)|f$ implies $\text{Norm}(v_1*v_2)=G_i(x)^2|\text{Norm}(f)$ and this contradicts the assumption that the Norm(f) was square free. Therefore the $\gcd(f,G_i(x))$ must be irreducible for all i. ∎

The only missing step in the previously outlined factoring procedure is finding a linear substitution that makes Norm(f) square free. We claim that Norm(f(x+sα)) is square free for some s in k. We will prove this in two stages, first for f(x) a polynomial over k and then extend the result to polynomials over $k(\alpha)$.

Theorem 2.3: *If f(x) is a square free polynomial with coefficients in k, then there are only a finite number of s in k such that Norm(f(x-sα)) has a multiple root.*

Proof: Let the roots of f(x) be $\beta_1,\beta_2,\ldots,\beta_m$, all distinct; then the roots of $f(x-s\alpha_j)$ are $\beta_1+s\alpha_j,\ldots,\beta_m+s\alpha_j$. Let G(x) = Norm(f(x-sα)) = $\text{product}_i f(x-s\alpha_i)$. Thus the roots of G are $s\alpha_k+\beta_i$ for $k\le n, i\le m$. G can have a multiple root only if $s = (\beta_j-\beta_i)/(\alpha_k-\alpha_m)$ where $k\ne m$. Therefore there are only a finite number of such values. ∎

Lemma 2.4: *If $f(x,\alpha)$ is a square free polynomial with coefficients in $k(\alpha)$ then there exists a square free polynomial g(x) over k such that $f|g$.*

Proof: Let G(x) = Norm(f(x,α)), let $\Pi g_i(x)^i$ be a square free decomposition of G. Then g(x) = $\Pi g_i(x)$ is a polynomial over k. Since f is square free and we have only discarded the multiple factors of G(x), g(x) is divisible by f. ∎

Corollary 2.5: *If $f(x,\alpha)$ is a square free polynomial over $k(\alpha)$ then there are only a finite number of s in k such that Norm(f(x-sα)) has a multiple root.*

Proof: Let g(x) be a polynomial over k as in the lemma. By theorem 2.3 there are only a finite number of s such that Norm(g(x-sα)) has multiple roots. But f|g implies Norm(f(x-sα,α)) divides Norm(g(x-sα)) and thus any multiple root of the former is a multiple root of the latter. ∎

We are now ready to present the entire factoring algorithm, but we will split off the norm computation as a subroutine for later use by other procedures.

Algorithm sqfr_norm

input: f(x,α) a square free polynomial over $k(\alpha)$

output: a positive integer s, g(x,α)=f(x-sα,α), R(x)=Norm(g(x,α)) a square free polynomial over k.

(1) s=0, g(x,α)=f(x,α) [initialize]

(2) R(x)=resultant($P_\alpha(y)$,g(x,y),y) [Norm,(resultant taken with respect to y)]

(3) If degree(gcd(R(x),R'(x))=0 then return (s,g,R) [sqfr check]

(4) s=s+1, g(x,α)=g(x-α,α), go to (2) ∎

Algorithm alg_factor

input: f(x,α) a square free polynomial over $k(\alpha)$

output: a list of irreducible factors over $k(\alpha)$

(1) (s,g,R)=sqfr_norm(f(x,α))

(2) L=factor(R(x)) [over k, returns list of factors]

(3) If length(L)=1 then return (f) [original poly was irreducible]

(4) For each $h_i(x)$ in L Do

(4.1) $h_i(x,\alpha) = \gcd(h_i(x),g(x,\alpha))$

(4.2) $g(x,\alpha) = g(x,\alpha)/h_i(x,\alpha)$ [performed over $k(\alpha)$]

(4.3) $h_i(x,\alpha) = h_i(x+s\alpha,\alpha)$ [undoes linear transformation]

(5) return (L) ∎

This factoring algorithm is similar to the one presented in van der Waerden [11 pp. 136-7] but computationally more efficient. If one wants to factor a univariate polynomial of degree d over an extension field of degree n, van der Waerden's approach requires computing a norm which is bivariate of degree nd in each variable and then factoring it over k. The algorithm presented above leads to the computation and factoring of a univariate norm of degree nd. It appears we have the additional cost of finding a linear transformation which makes the norm square free. However, the first step in factoring a bivariate polynomial over the k is to find a substitution for one of the variables which makes the result square free. Thus there is no actual additional cost and this algorithm is always superior to van der Waerden's.

In [13] Paul Wang gives another algorithm for factorization over algebraic number fields. His approach is an extension of his algorithm for factoring over the integers [12]. Wang's algorithm utilized van der Waerden's technique when the minimal polynomial for the algebraic number factored over primes, e.g. x^4+1. He now uses our improved algorithm in this case. In other cases, Wang's algorithm appears somewhat faster than ours, but his is restricted to algebraic numbers. Our algorithm can also be used on algebraic functions and thus has the advantages of increased generality as well as simplicity. We expect to analyze the computing times in both algorithms in the near future.

3. Primitive Elements

Next we will present an algorithm for computing a primitive element for a tower of extension fields. All of the algorithms we have presented have assumed that the extension field we operate in can be described by the adjunction of a single element to our base field k. If our current extension field is $k(\alpha)$ and β is algebraic over $k(\alpha)$ with minimal polynomial $Q_\beta(x,\alpha)$ then $k(\alpha,\beta)$ is the field obtained by adjoining β to $k(\alpha)$. We seek some γ which is algebraic over k such that $k(\gamma)=k(\alpha,\beta)$. The following theorem will prove very useful.

Lemma 3.1: *Let $P_\alpha(x)$ be the minimal polynomial for α over k and β be a root of $Q(x,\alpha)$, a square free polynomial. If $Norm_{[k(\alpha)/k]}(Q_\beta(x))$ is square free then $\gcd(P_\alpha(x),Q(\beta,x))$ is linear.*

Proof: α is clearly a root of both $P_\alpha(x)$ and $Q(\beta,x)$ since $Q(\beta,\alpha)=0$. Let the other roots of $P(x)$ be α_j for j=2,...,n. If $Q(\beta,\alpha_j)=0$ then β is a root of both $Q(x,\alpha)$ and $Q(x,\alpha_j)$, but Norm(Q) is $\Pi Q(x,\alpha_i)$ and then β is a multiple root of the norm. Since the norm is square free this cannot happen and the only common root of $P(x)$ and $Q(\beta,x)$ is α. Therefore the $\gcd(P(x),Q(\beta,x))$ is linear. ∎

Theorem 3.2: *Let $Q_\beta(x,\alpha)$ be the minimal polynomial for β over $k(\alpha)$ and $P_\alpha(x)$ be the minimal polynomial for α over k. If $Norm_{[k(\alpha)/k]}(Q_\beta(x))$ is square free then $k(\alpha,\beta) = k(\beta)$.*

Proof: We only need to show that α is representable in $k(\beta)$. By the lemma the $\gcd(Q(\beta,x),P(x))=x-c$, i.e. is linear. c is the only common root of $Q(\beta,x)$ and $P(x)$, so $c=\alpha$. But $Q(\beta,x)$ and $P(x)$ are both polynomials over $k(\beta)$ and so their gcd is over $k(\beta)$. Thus $\alpha=c$ is in $k(\beta)$. ∎

Given an arbitrary β the norm($Q_\beta(x,\alpha)$) may not be square free, but algorithm sqfr_norm will find an integer s and a polynomial $g(x,\alpha)$ such that $R(x) = Norm(g(x,\alpha))$ is square free. Since $Q(x,\alpha)$ is irreducible over $k(\alpha)$ and $s\alpha$ is in $k(\alpha)$, $g(x,\alpha) = Q(x-s\alpha,\alpha)$ is irreducible over $k(\alpha)$. If we let γ be a root of $g(x,\alpha)$ over $k(\alpha)$ and thus a root of $R(x)$ over k then by theorem 3.2 $k(\alpha,\gamma) = k(\gamma)$. But $\gamma = \beta+s\alpha$ so $k(\alpha,\gamma) = k(\alpha,\beta)$. Thus γ is the primitive element we were looking for and $R(x)$ is its minimal polynomial over k.

The algorithm we present for calculating primitive elements is essentially the same as that presented by Loos [4]. We differ in allowing the minimal polynomial for β to have coefficients over $k(\alpha)$ instead of requiring it to be over k. Loos does not guarantee, as we do, that the polynomial returned be irreducible. To achieve this result we must require that the minimal polynomial for β be irreducible over $k(\alpha)$ not just irreducible over k. In fact, if we examine our algebraic factoring algorithm, we can determine the conditions under which the resultant of two irreducible (over k) polynomials will factor. This will happen if and only if each of the polynomials factors over the extension field determined by the other polynomial. By symmetry, it is sufficient if one of the polynomials factors over the other's extension field.

Algorithm primitive_element

input: $P_\alpha(x)$ the minimum polynomial for α over k

$Q_\beta(x,\alpha)$ the minimum polynomial for β over $k(\alpha)$

output: R(x) the minimum polynomial for γ over k where $k(\alpha,\beta) = k(\gamma)$.

$A(\gamma)$ is a representation of α in $k(\gamma)$

$B(\gamma)$ is a representation of β in $k(\gamma)$

(1) (s,g,R) = sqfr_norm(Q(x,α),P(x))

(2) α=linsolve(gcd(g(γ,x),P(x))) [arithmetic over $k(\gamma)$ where γ denotes a root of R(x), linsolve($ax-b$) returns b/a]

(3) $\beta=\gamma-s\alpha$

(4) return (R(x),γ,α,β) ∎

4. Splitting Fields

The last algorithms to be presented in this section calculate splitting fields. We restrict our considerations to irreducible polynomials since the composite field from many such extensions can be found by repeated application of the primitive_element algorithm. Our basic approach will be to alternately adjoin a root of an irreducible factor of the polynomial to the current extension field and then refactor the polynomial in the new extension field. As linear factors are discovered they are put on a separate list and their coefficients are updated as the extension field changes. If n is the degree of the original polynomial, in the worst case $n-1$ iterations will occur and the primitive element for the splitting field will be of degree n!.

The norm computation at step 4.1 serves a dual purpose. It is used to find a minimal polynomial for the new extension field at step 4.3.2, but it is also the first stage for the algebraic factoring performed in steps 4.2 and 4.3. Thus in the context of splitting field calculations, our factoring algorithm becomes even more efficient.

Algorithm split_field

input: P(x) a polynomial irreducible over k

output: $R_\gamma(x)$ the defining polynomial for the splitting field of P(x) and a list of the roots of P(x) over $k(\gamma)$.

(1) roots = [], polys = [P(x)]

(2) minpoly = P(x), newminpol = P(x), index = 1, $\beta=\gamma$ [γ is a root of minpoly]

(3) replace polys[index] by polys[index]/(x-β), add β to roots, Newfactors = [], k = 1

(4) for each $P_i(x)$ in polys do

(4.1) (g,s,R) = sqfrnorm(P_i(x),minpoly)

(4.2) L = factor (R(x))

(4.3) for each $Q_j(x)$ in L do

(4.3.1) f(x,γ) = gcd(g(x,γ),Q_j(x))

[f is an irred. factor of P_i in $k(\gamma)$]

(4.3.2) if Deg(Q_i)>Deg(newminpol) then do

(4.3.1.1) newminpol = Q_j(x), index=k, new_s=s, Bpoly(x,γ) = f(x,γ)

(4.3.3) g(x,γ) = g(x,γ)/f(x,γ)

(4.3.4) f(x,γ) = f(x+sγ,γ)

(4.3.5) If Deg(f(x,γ)) = 1

(4.3.5.1) then add linsolve(f(x,γ)) to roots

(4.3.5.2) else add f(x,γ) to Newfactors, k=k+1

[let new_γ be a root of newminpol, now we operate in k(new_γ)]

(5) α = linsolve(gcd(minpoly,Bpoly(new_γ,x)))

(6) β = new_γ-new_sα

(7) subst α for γ in roots [update for new extension]

(8) If Newfactors = [] then return (newminpol,roots)

(9) subst α for γ in Newfactors

(10) polys = Newfactors, minpoly = newminpol, γ = new_γ, go to 3. ∎

For comparison, we now present a simpler version of the above algorithm which avoids performing any factoring. If the splitting field is actually of degree n! this approach will actually be faster since the attempt at factorization in step 4.2 would always fail and thus be a waste of time. The essence of the splitting field calculation is repeated primitive

element calculations. In the algorithm above factorization is attempted in the hope that the degree of the splitting field is much less than n!. The algorithm below always returns a polynomial of degree n! known as the resolvent [2]. The irreducible factors of this polynomial over k are all of the same degree, all normal, and are all defining polynomials for the splitting field of P(x).

Algorithm resolvent

input: P(x) a polynomial irreducible over k

output: R(x) a polynomial of degree n! such that any irreducible factor of R defines a splitting field for P

(1) minpoly = P(x), $\beta=\gamma$ [γ is a root of minpoly]

(2) $P(x,\gamma) = P(x)/(x-\beta)$

(3) If degree(P)=0 then Return minpoly

(4) (g,s,R) = sqfrnorm(P(x,γ),minpoly)

[Let new_γ be a root of R(x), now operate in k(new_γ)]

(5) α = linsolve(gcd(minpoly,P(new_γ,x)))

(6) β = new_γ - sα

(7) subst α for γ in P(x,γ)

(8) minpoly = R(x), γ = new_γ, go to 2. ∎

5. Rational Function Integration

We now turn to an application of the algorithms presented in the previous sections. The problem of the symbolic integration of rational functions in the context of algebraic manipulation has been investigated by Manove et al [8], Moses [9], Horowitz [3], Tobey [10], and Mack [6]. By Liouville's theorem the integral of a rational function can be expressed as a rational function plus a sum of complex constants times logs of rational functions.

$$\int R(x)dx = v_0(x) + \sum c_i \log(v_i(x))$$

Algorithms for obtaining the rational part of the integral are well known and reasonably efficient [3], [6], but as far as the author knows, no one has actually presented practical and relatively general algorithms for obtaining the transcendental part. Horowitz limited his investigation to algorithms for obtaining the rational part. Manove and by extension Moses' SIN factored the denominator over the integers and obtained logarithmic terms if the factors were linear or quadratics. Tobey examined the transcendental problem and concluded "There is no generally valid algebraic algorithm for obtaining in a symbolic form the transcendental part of the integral of a rational function." He did however present some algorithms for obtaining the transcendental part in special cases. He also presented as an unsolved problem the problem of obtaining the least degree extension field in which the integration can be done.

Starting where Horowitz leaves off, we are interested in integrating a rational function S(x)/T(x) where T(x) is square free and degree(S(x)) < degree(T(x)). If we were willing to operate in the splitting field of the denominator then we could perform a partial fraction expansion to get $S(x)/T(x) = \sum c_i/(x-\theta_i)$ where each c_i is an element of the splitting field and θ_i is a root of the denominator. In fact c_i is just the residue of R(x) at θ_i. This would lead to an expression for the integral, but could require operating in an extension field of very high degree. Thus for the sake of efficiency and to promote a more intelligible result, we propose to find the minimum degree extension field in which the result can be expressed. We claim that any field which contains all the residues is sufficient for expressing the integral. This is a significant result since the residues may be contained in a field of much lower degree than the splitting field of the denominator.

Our approach to this problem is first to constructively find the extension field E determined by all of the residues of the integrand. Then we factor the denominator of the integrand over this field and perform a partial fraction decomposition. This breaks the original integral into a sum of integrals of proper rational functions where each denominator is irreducible over E. We claim that each integral in the sum can be expressed as a single log term.

First we will find a simple expression for the residues in terms of the roots of the denominator T(x). Let θ be a root of the square free polynomial T(x). Since θ must then be a simple root, the residue of S(x)/T(x) at θ is $S(\theta)/T'(\theta)$. Since T(x) is square free, the gcd(T(x),T'(x))=1. Thus by the extended Euclidean algorithm we can find polynomials A(x) and B(x) over k such that A(x) T(x) + B(x) T'(x) = S(x), with the degree(B(x)) < degree(T(x)). $T(\theta) = 0$ implies $B(\theta) = S(\theta)/T'(\theta)$. Since θ was an arbitrary root of T(x) we have established the following:

Lemma 5.1: *Let S(x) and T(x) be polynomials over* k, *where T(x) is square free. Then there exists a polynomial B(x) over* k *such that for any root θ of T(x) the residue of S(x)/T(x) at θ is $B(\theta)$.*

Theorem 5.2: *Let S(x)/T(x) be a quotient of polynomials where T(x) is irreducible over some ground field k. If all the residues are contained in k then all the nonzero residues are equal.*

Proof: Let B(x) be the polynomial described in the lemma. Let the roots of T(x) be θ_i for i=1,...,n. We will operate in the splitting field E of T(x). Since T(x) is irreducible, its Galois group G(E/k) is transitive and there is an automorphism ϕ_j sending θ_1 to θ_j for each j and leaving k unchanged. Let $c = B(\theta_1)$, the residue at θ_1. If we apply ϕ_j to this equation, the left side is unchanged since c is in k and the right side is mapped to $B(\theta_j)$ since the coefficients of B(x) are in k. Thus $c = B(\theta_j)$ for each j and all the residues are equal. ∎

Corollary 5.3: *With S(x) and T(x) as in Theorem 5.2, $\int S(x)/T(x)dx$ is expressible over k.*

Proof: Let c be the common residue of the θ_j, then the integrand has the partial fraction decomposition, $S(x)/T(x) = \sum c/(x-\theta_j)$. This integrates to the following:

$$\int S(x)/T(x)dx = \sum c\log(x-\theta_i) = c \log \Pi(x-\theta_i) = c \log T(x) \ . \ \blacksquare$$

All that remains is to transform the integrand so that the corollary applies. If θ is any root of the denominator T(x) then $B(\theta)$ is the residue of $S(x)/T(x)$ at γ, therefore the resultant of x-B(y) and T(y) with respect to y is a polynomial whose roots are precisely the residues of $S(x)/T(x)$. Using the algorithms presented in the last section we can compute the splitting field of this polynomial and therefore the least degree extension which contains all the residues. Then we factor the denominator of the integrand over this extension field. If we performed a partial fraction decomposition on the resulting rational function, each term in the sum would satisfy the hypotheses of the corollary and thus be directly integrable. But the actual computation of the partial fraction decomposition is unnecessary. Since we know the form of the result as $\sum c_i \log(f_i(x,\gamma))$ where the f_i's are the irreducible factors of the denominator, all we need to be able to do is compute each c_i. But c_i is B(x) evaluated at any root of $f_i(x,\gamma)$. The resultant(B(x),$f_i(x,\gamma)$,x) is the product of B(x) evaluated at each of the roots of f and thus equals c_i^k where k is the degree of f. Therefore g(y) = resultant(y-B(x),$f_i(x,\gamma)$,x) is $(y-c_i)^k$, and $y-c_i$ is g(x)/Gcd(g(x),g'(x)).

Algorithm ratint

input: T(x) a square free polynomial, S(x) a polynomial of lower degree than T(x)

output: R(x) the minimal polynomial for the splitting field of the residues, γ such that $R(\gamma) = 0$, and $I(x,\gamma)$ the integral expressed in terms of γ.

(1) (A,B) = Extended_Euclidean(T(x),T'(x),S(x))

(2) R(x) = split_field (resultant(x-B(y),T(y),y))

(3) L = alg_factor(T(x),R(x),γ)

(4) L = map(Int_log,L) [applies function Int-log to each element in L]

(5) Return(R(x),γ,sum(L)) $\blacksquare$

Algorithm int-log

input: D(x,γ) an irreducible polynomial over $k(\gamma)$

output: c log D(x,γ)

(1) c(y) = resultant(y-B(x),D(x,γ),x)

(2) c = linsolve(c(y)/Gcd(c(y),c'(y))) [common residue expressed in terms of γ.]

(3) return (c log D(x,γ)) $\blacksquare$

On page III-10 of his thesis, Tobey presents a rational function which he demonstrates is integrable over Q(sqrt(2)). He asks how one determines a priori the extension of least degree in which the integral can be expressed. Using the MACSYMA [7] system and the ideas presented in this section, we solve his problem below:

```
(C1)'INTEGRATE(S(X)/T(X),X);/* THIS IS TOBEY'S INTEGRAL */
(D1)

  /      13       8      7      6      3      2
  [ 7 X   + 10 X  + 4 X  - 7 X  - 4 X  - 4 X  + 3 X + 3
  I ------------------------------------------------- DX
  ]      14      8      7      4      3    2
  /   X   - 2 X  - 2 X  - 2 X  - 4 X  - X  + 2 X + 1

(C2) (ALGEBRAIC:TRUE,TELLRAT(T(C)))$ /* LET C BE A ROOT OF
                                          T(X) */

(C3) B(C):=''(RATSIMP(N(C)/DIFF(D(C),C))); /* B(X) IS THE
                 POLY COMPUTED IN STEP 1 OF RATINT */
(D3) B(C) :=

     12    11   10    9    8    7    6      2
    C   - C  + C   - C  + C  - C  - C  - 2 C  - 2 C + 2
    ---------------------------------------------------
                                2

(C4) RESULTANT(Y-B(X),D(X),X);
                14            13            12            11
(D4) 16384 Y   - 114688 Y   + 315392 Y   - 401408 Y

            10            9            8            7            6
+ 164864 Y   + 121856 Y  - 109312 Y  - 23552 Y  + 27328 Y

         5          4          3          2
+ 7616 Y  - 2576 Y  - 1568 Y  - 308 Y  - 28 Y - 1

(C5) SQFR(%); /* SQUARE FREE DECOMPOSITION ISN'T NECESSARY
      BUT MAKES STRUCTURE MORE EVIDENT IN THIS EXAMPLE */
                                2            7
(D5)                    (4 Y  - 4 Y - 1)

(C6) MP(X):=''(SUBST(X/2,Y,PART(%,1))); /* THIS IS THE
       MINIMAL POLY. FOR THE RESIDUES, MONICIZED FOR
       EFFICIENCY. */
                                2
(D6)                 MP(X) := X  - 2 X - 1

(C7) FACTOR(T(X),MP(ALG)); /* ALGEBRAIC FACTORIZATION OF
       DENOMINATOR OF INTEGRAND */
       7                2                7                2
(D7) (X  + (1 - ALG) X  - ALG X - 1) (X  - (ALG - 1) X

                                          + (ALG - 2) X - 1)

(C8) (F1:PART(%,1),F2:PART(%,2)); /* FI IS ITH FACTOR */
                7                2
(D8)          X  + (ALG - 1) X  + (ALG - 2) X - 1

(C9) TELLRAT(MP(ALG))$ /* NEXT WE CALCULATE RESIDUES OVER
                                  k(ALG) */

(C10) RESULTANT(Y-B(X),F1,X);
            7              6                      5
(D10) 128 Y  - 448 ALG Y  + (1344 ALG + 672) Y

                          4                      3
 + (- 2800 ALG - 1120) Y  + (3360 ALG + 1400) Y

                          2
 + (- 2436 ALG - 1008) Y  + (980 ALG + 406) Y

 - 169 ALG - 70
```

```
(C11) SOLVE(%,Y);
SOLUTION

                                ALG
(E11)                      Y = ---
                                 2
MULTIPLICITY 7
(D11)                       [E11]

(C12) C1:EV(Y,%);/* C1 IS THE RESIDUE AT ANY ROOT OF F1 */

                               ALG
(D12)                          ---
                                2

(C13) SOLVE(RESULTANT(Y-B(X),F2,X),Y);
SOLUTION

                                ALG - 2
(E13)                    Y = - -------
                                   2
MULTIPLICITY 7
(D13)                       [E13]

(C14) C2:EV(Y,%); /* C2 IS RESIDUE AT ANY ROOT OF F2 */
                               ALG - 2
(D14)                        - -------
                                  2

(C15) C1*LOG(F1)+C2*LOG(F2); /* FINALLY WE CAN EXPRESS
                                 THE INTEGRAL */
                 7                2
      ALG LOG(X  + (1 - ALG) X  - ALG X - 1)
(D15) ------------------------------------
                        2

                          7                2
        (ALG - 2) LOG(X  + (ALG - 1) X  + (ALG - 2) X - 1)
      - ------------------------------------------------
                                 2

(C16) (ALGEBRAIC:FALSE,SOLVE(MP(ALG))); /* SINCE MP(X) IS
      QUADRATIC, WE CAN EXPRESS THE ANSWER IN RADICALS */
SOLUTION

(E16)                  ALG = 1 - SQRT(2)

(E17)                  ALG = SQRT(2) + 1
(D17)                      [E16, E17]

(C18) EV(D15,E17);
(D18)

                    7                2
(SQRT(2) + 1) LOG(X  - SQRT(2) X  - (SQRT(2) + 1) X - 1)
-------------------------------------------------------
                          2

                      7                2
  (SQRT(2) - 1) LOG(X  + SQRT(2) X  + (SQRT(2) - 1) X - 1)
- -------------------------------------------------------
                            2
```

We have shown that any integral can be expressed over the field generated by the residues of the integrand. Now we must justify our claim that this field is the minimal one, i.e. we will show that if an integral can be expressed over a field k, then all the residues of the integrand lie in k.

Let $v_0(x) + \sum c_i \log(v_i(x))$ be the integral of a rational function, and let all the coefficients be contained in some field k. Then the integrand can be expressed as $v'_0 + \sum c_i v'_i / v_i$. The residue of v'_0 is zero everywhere, while the residue of v'/v is always a rational integer. Thus all the residues of the integrand can be expressed as integer linear combinations of the c_i and thus must lie in k. Combined with our earlier result, we have that the field determined by the residues is both necessary and sufficient for expressing the integral.

6. Extensions and Comments

The algorithms presented in this paper were designed to operate over an arbitrary base field. If we are interested in factoring univariate polynomials over algebraic number fields then we let our base field be the rational numbers. Given the capability to factor multivariate polynomial norms over Q as in [12], we can extend to factoring multivariate polynomials over algebraic number fields. If we allow our minimal polynomials to have polynomial coefficients then we can factor polynomials over algebraic function fields.

As algebraic manipulation systems expand their problem domains, the need for performing operations with quantities satisfying algebraic relationships will increase. The basic arithmetic operations can be performed by merely using the side relations to keep the expressions reduced. We have extended factoring to these domains by mapping the problem to a simpler domain while still preserving its structure. Then we were able to lift the factorization back to the original expression. That finding such unramified morphisms can lead to efficient algorithms for algebraic manipulation has been amply demonstrated by the recent development of modular and p-adic techniques [15].

I would like to thank Joel Moses, Richard Zippel, and David Barton for many enlightening discussions. Work herein was supported in part by the United States Energy Research and Development Administration contract E(11-1)-3070.

References

[1] Collins, G.E., "The Calculation of Multivariate Polynomial Resultants", *JACM*, vol. 18, no. 4, Oct. 1971, pp. 515-532.

[2] Gaal, L., *Classical Galois Theory with Examples*, Markham, Chicago, 1971, reprinted by Chelsea, New York.

[3] Horowitz, E.,*Algorithms for Symbolic Integration of Rational Functions*, Ph.D. Thesis, U. of Wisconsin, 1970.

[4] Loos, R. G. K., "A Constructive Approach to Algebraic Numbers", Computer Science Dept., Stanford University, Palo Alto, Calif.

[5] MacDuffee, C., *An Introduction to Abstract Algebra*, Dover, 1966.

[6] Mack, D., *On Rational Integration*, Computer Science Dept., Utah Univ., UCP-38, 1975.

[7] *MACSYMA Reference Manual*. Mathlab Group, Project MAC, M.I.T., Cambridge, Mass., November 1975.

[8] Manove, M., Bloom, S., and Engelman, C., "Rational functions in MATHLAB", *Proc. IFIP Conf. on Symbolic Manipulation Languages*, Pisa, Italy, 1968.

[9] Moses, J., "Symbolic Integration: The Stormy Decade", *Communications of the ACM*, vol 14, no 8, pp. 548-560, 1971.

[10] Tobey, R.G., *Algorithms for Antidifferentiation of Rational Functions*, Ph.D. Thesis, Harvard, 1967.

[11] van der Waerden, B.L., *Modern Algebra*, vol 1, tr. Fred Blum, Frederick Ungar Publishing Co., New York, 1953.

[12] Wang, P.S. and Rothschild, L.P., "Factoring Multivariate Polynomials Over the Intgers," *Mathematics of Computation*, vol 29, no. 131, pp 935-950, 1975.

[13] Wang, P.S., "Factoring Multivariate Polynomials over Algebraic Number Fields", *Mathematics of Computation*, vol. 30, no. 134, April 1976.

[14] Weyl, Hermann, *Algebraic Theory of Numbers*, Princeton University Press, 1940.

[15] Yun, D.Y.Y., *The Hensel Lemma in Symbolic Manipulation*, Ph.D. Thesis, M.I.T., MAC TR-138, 1974.

Some Problems about Polynomials

by Maurice Mignotte
Université Louis Pasteur Strasbourg

Abstract: We consider the following topics: upper bounds for the roots of a polynomial, the minimum distance between distinct roots, Bernouilli's method, and factorization over the integers. Some problems are solved, others are left open.

0. Notations.

Let $P = a_0 X^d + \dots + a_d$ be a monic polynomial with complex coefficients. The height of P is $\|P\| = \max|a_i|$. We put

$$\|P\|_p = (|a_0|^p + \dots + |a_d|^p)^{1/p}, \quad p \geq 1 .$$

This is an increasing function of p. Let $z_1, \dots, z_d$ be the roots of P. We put

$$M(P) = \max(|z_1|, 1) \dots \max(|z_d|, 1) .$$

1. Upper bounds for the roots of a polynomial.

It is well-known, and easily proved, that each root of P satisfies

(1) $\quad |z_i| \leq \|P\| + 1 .$

In fact, many refinements of this inequality are known (see [8]). But, from a certain point of view, inequality (1) is best possible : it is not true in general if we replace $\|P\|$ by $\|P\|^c$ for some positive fixed number c, $c < 1$.

It is often necessary (see § 4) to obtain an upper bound for the product $(|z_1|+1)\dots(|z_d|+1)$ or some other expression of the same kind. From (1) we get

(2) $\quad (|z_1|+1)\dots(|z_d|+1) \leq (\|P\|+2)^d .$

In this inequality the dependance in $\|P\|$ is very bad. In fact it is known, but perhaps not well-know, that it is possible to obtain an upper bound which is linear in $\|P\|$. The best known result in this direction is the following.

Théorème 1.- With the notations of the beginning, we have $M(P) \leq \|P\|_2$. (This inequality has been found and forgotten several times. It occurs in a paper of Landau [6], the proof uses Parseval formula. Two purely algebraic proofs can be found in [4] and [9].)

- It is interesting to note that the quadratic norm is best possible in theorem 1 : for fixed $p > 2$, the upper bound $M(P) \leq \|P\|_p$ is not true in general (see [9]). Now if we apply theorem 1 to the product considered in (2) we get the much better result

(3) $\quad (|z_1|+1)\dots(|z_d|+1) \leq c_d \|P\| ,$

where $c_d = 2^d \sqrt{d+1}$.

(Proof : Use the inequalities $\|P\|_2 \leq \sqrt{(d+1)}\, \|P\|$ and $|z|+1 \leq 2 \max(1, |z|)$.)

Our first question is :

Problem 1 : For each degree d, what is the best constant c_d such that (3) is true for any polynomial of degree d ?

2. On the minimum root separation of a polynomial.

A recent paper of Collins and Horowitz [3] brought my attention on this problem. Let $\mathrm{sep}(P)$ be the minimum distance between distinct roots of P. For many algorithms it is necessary to know a lower bound for $\mathrm{sep}(P)$. Moreover, the complexity of the algorithms which isolate the roots of a polynomial is bounded below by power of $\log(1/\mathrm{sep}(P))$ (see [3] for a more detailed discussion). We shall give the best lower bound which seems to be published and see how useful is theorem 1.

Théorème 2.- If P is a polynomial with discriminant $D(P)$ then

$$\mathrm{sep}(P) > \sqrt{3}\, d^{-(d+2)/2} |D(P)|^{1/2} \|P\|_2^{-d+1} .$$

> Mahler [7] proved that $\mathrm{sep}(P)$ satisfies

$$\mathrm{sep}(P) > \sqrt{3}\, d^{-(d+2)/2} |D(P)|^{1/2} M(P)^{-d+1} .$$

The result follows from theorem 1.<

In [2] the following problem was considered. Put $L(d, H) = \mathrm{Min}\{\mathrm{sep}(P) ; P \in \mathbb{Z}[X]$ without multiple roots, $\deg(P) = d$, $\|P\| \leq H\}$. It is easy to prove that $H^{-1} \ll L(2, H) \ll H^{-1}$ (where $\ll$ is Vinogradof'snotation : $f(x) \ll g(x)$ means that there is some positive constant C such that $f(x) \leq C\, g(x)$). Collins and Horowith proved that $H^{-3} \ll L(3, H) \ll H^{-1}$. In [10] we obtained $H^{-2} \ll L(3, H) \ll H^{-2}$. Now it was natural to consider

$$L(d) = \lim_{H \to \infty} \sup(-\mathrm{Log}(L(d, H))/\mathrm{Log}\, H) .$$

Theorem 2 gives $L(d) \leq d-1$. Using a result of W. M. Schmidt ([12], th. 7.I) about approximation of algebraic numbers by algebraic numbers of smaller degree we proved the following result.

Théorème 3.- We have $\lceil (d+1)/2 \rceil \leq L(d) \leq d-1$.

Collins [2] suggests a modification : "Suppose that in the definition of $L(d, H)$ we now require that P be irreducible. This has practical significance because we might begin the calculation of the roots of a polynomial by factoring it into irreducible factors...". Thus we consider the following variants of $L(d)$: $L_0(d)$ (respectively $L_1(d)$) where we take only irreducible (respectively reducible) polynomials over $\mathbb{Z}$. We have now two different problems.

Problem 2 : Give a lower bound for $L_0(d)$. (We are able to prove $L_0(d) \geq 1$. Is it possible to do better ?)

Problem 3 : Find the best upper bound for $L_1(d)$.

3. On Bernouilli's method.

Bernouilli's method to calculate the roots of complex polynomials is described in [5] . And Henrici asked the following question.

Problem 4 : Suppose a polynomial has exactly three dominant zeroes, one real, two complex conjugate. Devise a modification of Bernouilli's method that deals with this situation.

In [10] we give a solution to this problem and show that Bernouilli's method always enables us to calculate $\max|z_i|$. Now there is another problem.

Problem 5 : Suppose a polynomial has exactly two pairs of complex conjugate dominant zeroes. Devise a modification of Bernouilli's methode that deals with this situation.

It is worthy to note that Turan [13] shows that it is possible to modify Graeffe's method to obtain a universal algorithm for computing a root of an arbitrary polynomial. This paper uses lower bounds such as

$$\max_{1\le i\le n}((t_1^i+\ldots+t_n^i)/n)^{1/i} > 1/5 \quad \text{if} \quad \max|t_i| = 1.$$

Such a result may be useful to solve problem 5 . But this result leads us to consider the following question.

Problem 6 : If $t_1,\ldots,t_n$ are algebraic numbers of modulus one, find a non trivial function $f(n)$ such that

$$t_1^k+\ldots+t_n^k \neq 0 \text{ implies } |t_1^k+\ldots+t_n^k| > f(k) .$$

Using a result of Baker [1] about linear forms in logarithms, I proved [11] that if z_1 is a non real algebraic number then

$$z_1^k+\bar{z}_1^k \neq 0 \text{ implies } |z_1^k+\bar{z}_1^k| > C_1 n^{-C_2}$$

where C_1 and C_2 are effectively computable positive constants. The same result is true for $1+z_1^k+\bar{z}_1^k$ and $z_1^k+\bar{z}_1^k+z_2^k+\bar{z}_2^k$, where z_2 is an algebraic number. But the general problem seems to be very difficult.

4. Factoring polynomials over the integers.

We suppose now that P has integer coefficients. Let Q be an integer polynomial of degree d' which divides P . We can suppose that $z_1,\ldots,z_{d'}$ are the roots of Q . It is clear that $\|Q\|$ satisfies $\|Q\| \le (|z_1|+1)\ldots(|z_{d'}|+1)$. Thus (1) gives $\|Q\| \le (\|P\|+1)^{d'}$ whereas (3) implies the far better inequality

(4) $\quad \|Q\| \le C_{d'}\|P\|$, where $C_{d'} = 2^{d'}\sqrt{d'+1}$.

Such an inequality plays an important role in the algorithm of Zassenhaus [14] . See [9] for a discussion of this question. Problem 1 is similar to

Problem 7 : Find the best constant $C_{d'}$, such that (4) is true for any polynomial Q of degree d' which divides another polynomial P .

References.

[1] A. Baker. - A sharpening of the bounds for linear forms in logarithms, Acta Arith., 21, 1974, 117-129.

[2] G. E. Collins. - Letter of 17.2.1976.

[3] G. E. Collins, E. Horowitz. - The minimum root separation of a polynomial, Math. Comp. 28, n° 126, 1974, 589-597.

[4] V. Goncalves. - L'inégalité de W. Specht, Rev. Fac. de Ciências de Lisboa, 1, 1950, 167-171.

[5] P. Henrici. - Elements of Numerical Analysis, J. Wiley, New-York, 1964.

[6] E. Landau. - Sur quelques théorèmes de M. Petrovič relatifs aux zéros des fonctions analytiques, Bull. Soc. Math. France, 33, 1905, 251-261.

[7] K. Mahler. - An inequality for the discriminant of a polynomial, Michigan Math. J., 11, 1964, 257-262.

[8] M. Marden. - The geometry of the zeroes of a polynomial in a complex variable, Mathematical Surveys n°3, A. M. S., New-York, 1949.

[9] M. Mignotte. - An inequality about factors of polynomials, Math. Comp., 28, 1974, 1153-1157.

[10] M. Mignotte. - Sur la complexité de certains algorithmes où intervient la séparation des racines d'un polynôme (RAIRO, to appear).

[11] M. Mignotte. - Note sur la méthode de Bernouilli (Num. Math., to appear).

[12] W. M. Schmidt. - Approximation to algebraic numbers, Monographie n°19 de l'Enseignement Mathématique, Genève, 1972.

[13] P. Turan. - Power Sum Method and the approximate solution of algebraic equations, Math. Comp., 29, n°129, 1975, 311-318.

[14] H. Zassenhaus. - On Hensel factorization I, J. Number Theory, 1, 1969, 291-311.

Using Basis Computation to Determine Pseudo-Multiplicative Independence *

H. I. Epstein
Department of Mathematics
Boston College

Abstract

This paper describes a procedure for determining when a set of rational functions are pseudo-multiplicatively independent, i.e. when no non-trivial power product of the rational functions can be a constant. The method used is to derive a multiplicative basis of factors of the numerators and denominators of the rational functions; the basis is then used to derive a system of homogeneous linear equations which will have a non-trivial solution if and only if the rational functions are pseudo-multiplicatively dependent. Algorithms which find a basis for a set of Gaussian rational functions and the corresponding linear equations are presented in detail and bounds on their theoretical computing times are derived.

1. Introduction

Multiplicative independence is the multiplicative analog of linear independence. Rather than asking whether there are n_i not all zero such that $\sum_{i=1}^{m} c_i R_i = 0$, ask whether there are integers n_i not all zero such that $\prod_{i=1}^{m} R_i^{n_i} = 1$. By weakening this condition to require merely that $\prod_{i=1}^{m} R_i^{n_i}$ equals a constant, the question of whether $R_1,\ldots,R_m$ are pseudo-multiplicatively independent is introduced.

The question of pseudo-multiplicative independence occurs naturally when doing transcendental function arithmetic involving logarithms. For, in doing symbolic computation involving the logarithms $\log A_1,\ldots,\log A_n$ it is desirable to find an algebraically independent collection of logarithms, $\log R_1,\ldots,\log R_n$, with the property that each $\log A_i$ can be represented as a polynomial in $\log R_1,\ldots,\log R_n$. Each $\log R_i$ can then be treated as an independent variable and so each $\log A_i$ can be represented as a polynomial in these logarithmic variables. If the logarithms chosen as independent variables are not algebraically independent, then expressions equivalent to zero may not be recognized as such and expression size might grow unnecessarily large in the course of calculations. Thus it is important that the logarithmic variables be algebraically independent. Since a set of logarithms will be algebraically independent if their arguments are pseudo-multiplicatively independent, the notion of pseudo-multiplicative independence is useful in doing symbolic mathematical computations with logarithms.

In [FAT72] and [JOH71] it is implied that deciding whether the power product $\prod_{i=1}^{m} R_i^{n_i}$ is a constant is equivalent to deciding whether $\sum_{i=1}^{m} n_i R_i'/R_i = \sum_{i=1}^{m} n_i (\log R_i)'$ is zero when each R_i is a univariate rational function. Determining whether $R_1'/R_1,\ldots,R_m'/R_m$ are linearly independent when the R_i are rational functions could be done as follows. First, find Q, the least common multiple of the denominators of the quotients R_i'/R_i. Next, determine whether the polynomials $QR_1'/R_1',\ldots,QR_m'/R_m$ are linearly independent by solving a system of linear equations obtained by setting each numerical coefficient in the sum $\sum_{i=1}^{m} n_i QR_i'/R_i$ to 0. If each R_i is a rational function in r variables with numerator and

*This research was supported in part by National Science Foundation grant GJ-30125X.

denominator of degree $d > 0$ in each variable, this method could be used by applying it r times, once with ' replaced by each of the r different partial derivatives. This could yield a system of $r(md)^r$ equations in m unknowns. That, together with the potential for coefficient growth in computing Q, and the space required to store the polynomials QR_i'/R_i, $1 \le i \le m$, suggests considering another approach.

This paper considers one alternative approach. That approach is to find a basis consisting of polynomial factors of the numerators and denominators of the R_i in order to obtain a multiplicative basis for $\{R_1,\ldots,R_m\}$ using a method suggested by Collins in [COL74]. Each factor in the basis gives rise to a linear equation. The coefficients in each such equation are the multiplicities of the factor (from which the equation arose) as a factor of each R_i. Thus, these linear equations tend to have rather small coefficients since the largest multiplicity of any factor of a polynomial of degree d must be $\le d$. Furthermore, if the numerator and denominator of each R_i is an r-variate polynomial of degree $\le d$ in each variable, each R_i can contribute only 2dr factors to the basis. So, there will be at most 2mdr linear equations in m unknowns.

Algorithms similar to those presented below have been implemented as part of the Transcendental Function subsystem of the SAC-1 system. A detailed discussion of this implementation can be found in [EPS75].

Section 2 presents some basic definitions and notation used in discussing the algorithms. In section 3, the algorithms for determining whether a set of functions are pseudo-multiplicatively independent are presented. Most of those algorithms are concerned with finding a basis for a set of rational functions and finding a representation for the linear equations to which they give rise. Section 4 analyzes the theoretical computing times of the algorithms presented in section 3. In section 5 are some concluding remarks concerning possible modifications of the algorithms in section 3.

2. Basic Concepts

Throughout this paper U will denote a unique factorization domain (ufd). Let P be a polynomial in $U[x_1,\ldots,x_r]$. The degree of P in x_j, $1 \le j \le r$ will be the degree of P when regarded as an element of $U[x_1,\ldots,x_{j-1}, x_{j+1},\ldots,x_r][x_j]$. The degree of P, written deg(P), will denote the degree of P in x_r. Thus, if $\deg(P) = n$, $P(x_1,\ldots,x_r) = \sum_{i=0}^{n} p_i(x_1,\ldots x_{r-1})x_r^i$ and $p_n(x_1,\ldots,x_{r-1}) \neq 0$; p_n is called the leading coefficient of P, written ldcf(P). Any element of U is said to be a constant. A polynomial P in $U[x_1,\ldots,x_r]$ is said to be a constant if deg(P) = 0 and ldcf(P) is a constant.

Elements a, b in U are said to be associates, written $a \sim b$, provided that there is a unit u in U such that a = ub. An ample set for U is a set $A \subseteq U$ which contains exactly one associate of each element. When U is the domain of Gaussian integers, take the set of first quadrant Gaussian integers $= \{a+bi : a > 0, b \ge 0\} \cup \{0\}$ as the ample set. If A is an ample set for U, take $\{P : \mathrm{ldcf}(P) \in A\}$ as an ample set for U[x]. Thus, when P is a Gaussian polynomial, its ample associate, which will be donoted FQV(P), is a Gaussian polynomial whose leading coefficient is a first quadrant Gaussian integer.

Let A be an ample set for U. For a,b in U, gcd(a,b) shall denote the unique ample associate of the greatest common divisor of a and b. When $\gcd(a,b) \sim 1$, a and b are relatively prime. Since FQV(1) = 1, when U is the Gaussian integers, if a,b are relatively prime gcd(a,b) = 1.

If R is an r-variate rational function in $U(x_1,\ldots,x_r)$, there is a unique representation of R=P/Q where P and Q are relatively prime polynomials in $U(x_1,\ldots,x_r)$ and Q is ample. P is called the numerator of R, written num(R), and Q is called the denominator of R, written den(R). If num(R) and den(R) are constants, then R is called a constant.

Let $R_1,\ldots,R_m$ be rational functions in $U(x_1,\ldots,x_r)$. Then $R_1,\ldots,R_m$ are said to be multiplicatively independent provided that the only integers $n_1,\ldots,n_m$ such that $\prod_{i=1}^{m} R_i^{n_i} = 1$ are $n_1=n_2=\ldots=n_m=0$. $R_1,\ldots,R_m$ are pseudo-multiplicatively independent, abbreviated pmi, provided the only integers $n_1,\ldots n_m$ such that $\prod_{i=1}^{m} R_i^{n_i}$ is a constant are $n_1=n_2=\ldots=n_m=0$. In the next section an algorithm is presented to determine when a set of non-constant Gaussian rational functions are pmi.

3. Pseudo-multiplicative Independence and Basis Computation

A polynomial P in $U[x_1,\ldots,x_r]$ is squarefree if there is no non-constant polynomial Q such that Q^2 is a factor of P. Let R be a set of rational functions in $U(x_1,\ldots,x_r)$.

A basis for R is a set $B = \{B_1,\ldots,B_n\}$ of ample, squarefree, non-constant polynomials with the three properties

(1) If B_1, B_2 are distinct elements of B, then $\gcd(B_1,B_2) \sim 1$.

(2) If B_i is in B then B_i is a factor of $\text{num}(R_j)$ or $\text{den}(R_j)$ for some R_j in R.

(3) If R_j is in R, then there are rational integers $e_{1j}, e_{2j},\ldots,e_{nj}$ and an element u_j in U such that $R_j = u_j \prod_{i=1}^{n} B_i^{e_{ij}}$.

The list $(e_{1j},e_{2j},\ldots,e_{nj})$ is called the multiplicity sequence of B corresponding to R_j. The matrix

$$E = \begin{pmatrix} e_{11} & e_{12} & \cdots & e_{1,m} \\ e_{21} & e_{22} & \cdots & e_{2,m} \\ \vdots & & & \vdots \\ e_{n1} & e_{n2} & \cdots & e_{n,m} \end{pmatrix}$$

is called the multiplicity matrix of B with respect to R. The i-th row of the multiplicity matrix is called the occurrence sequence of B_i relative to R. Notice that e_{ij} is the number of times B_i occurs as a factor of R_j.

The algorithms presented below for determining whether a set of rational functions is pmi are based on Theorem 1. By applying a standard result of linear algebra to the homogeneous system EX=0, where E is a multiplicity matrix for some set R of rational functions, the following theorem determines whether R is pmi.

Theorem 1. Let $R = \{R_1,\ldots,R_m\}$ be a set of non-constant rational functions in $U(x_1,\ldots,x_r)$. Let B be a basis for R and let $E = \begin{pmatrix} e_{11} & \cdots & e_{1,m} \\ \vdots & & \vdots \\ e_{n1} & \cdots & e_{n,m} \end{pmatrix}$ be the multiplicity matrix for B with respect to R. Then R is pmi iff the only solution to the matrix equation

(*) $EX=0$

is the trivial solution (i.e., X is a column matrix of m zeros).

Proof. Since B is a basis for R and E is the multiplicity matrix for B with respect to R, for each j there is a constant u_j in U such that $R_j = u_j \prod_{i=1}^{n} B_i^{e_{ij}}$. Thus $\prod_{j=1}^{m} R_j^{n_j} = \prod_{j=1}^{m} (u_j \prod_{i=1}^{n} B_i^{e_{ij}})^{n_j} = \prod_{j=1}^{m} u_j^{n_j} \cdot \prod_{i=1}^{n} B_i^{s_i}$ where $s_i = \sum_{j=1}^{m} e_{ij} n_j$. Since each u_j is in U, $\prod_{j=1}^{m} u_j^{n_j}$ is in U. Thus, $\prod_{j=1}^{m} R_j^{n_j}$ is a constant whenever $\prod_{i=1}^{n} B_i^{s_i}$ is a constant. Now, if every $s_i = 0$, $\prod_{i=1}^{n} B_i^{s_i} = 1$. But, every $s_i = 0$ precisely when the column vector $(n_1,\ldots,n_m)$ is a solution to (*). Observe that since each e_{ij} in E is an integer, if there is a non-trivial solution to (*), there is a non-trivial integer solution. But if there is such a solution $(n_1,\ldots,n_m)$, $\prod_{j=1}^{m} R_j^{n_j}$ is a constant and R is not pmi.

Conversely, suppose that the only solution to (*) is X = 0. Assume, in order to derive a contradiction, that there are integers $n_1,\ldots n_m$, not all zero, such that $\prod_{j=1}^{m} R_j^{n_j}$ is constant. As above, when $\prod_{j=1}^{m} R_j^{n_j}$ is a constant so is $\prod_{i=1}^{n} B_i^{s_i}$. Since some $n_j \neq 0$, the column vector $(n_1,n_2,\ldots,n_m)$ cannot be a solution to (*). So, some $s_i \neq 0$. Without loss of generality suppose $s_1 \neq 0$. Thus $\prod_{i=2}^{t} B_i^{s_i} = uB_1^{-s_1}$ for some u in U. Therefore $s_1 > 0$ implies B_1 is a factor of $\text{num}(\prod_{i=2}^{t} B_i^{s_i})$ and $s_1 < 0$ implies B_1 is a factor of $\text{den}(\prod_{i=2}^{t} B_i^{s_i})$. Since $\gcd(B_1,B_i) \sim 1$ whenever $1 < i \leq n$, this is impossible. This contradiction proves the theorem. ∎

Theorem 1 reduces the problem of determining whether a set of rational functions are pmi to that of factoring the numerators and denominators of that set of rational functions in order to obtain a basis B and multiplicity matrix E and then solving the system of homogeneous linear equations represented by EX = 0.

Let U be a ufd with ample set A. Let $P = \sum_{i=0}^{n} p_i x_r^i$ be an r-variate polynomial in $U[x_1,\ldots,x_r]$ of deg n. If $P \neq 0$, the content of P, written cont(P), is $\gcd(p_0,p_1,\ldots,p_n)$; cont(0) = 0. When $P \neq 0$, the primitive part of P, written pp(P), is the ample associate of P/cont(P); pp(0) = 0. When cont(P) = 1, P is said to be primitive.

Suppose P is a primitive r-variate polynomial in $U[x_1,\ldots x_r]$. P may be written $P = u \prod_{j=1}^{k} P_j^{\,j}$ where u is in U and the P_j are ample, primitive, squarefree, pairwise relatively prime polynomials. The list $(P_1,\ldots,P_k)$ is called a squarefree factorization of P. Some P_j may be constant. If U is the Gaussian integers and $\deg(P_j) = 0$, $P_j = 1$ since it must be a first quadrant (because it is ample) unit (since P is primitive). However, the P_j which are not constant form a basis for {P} and the list of subscripts of those P_j form the column matrix which is the corresponding multiplicity matrix. The next theorem gives a bound on the number of non-constant factors in the squarefree factorization of a polynomial.

Theorem 2. Let $(P_1,\ldots,P_n)$ be the squarefree factorization of the primitive polynomial P where $\deg(P) = d > 0$. Suppose there are t non-constant polynomials among $P_1,\ldots,P_n$. Then $t < (2d)^{1/2}$.

Proof. Let $I = \{i: \deg(P_i) \neq 0\}$. Then $d=\deg(P)=\deg(\prod_{i\in I} P_i^{\,i}) = \sum_{i\in I} i \deg(P_i) \geq \sum_{i\in I} i \geq \sum_{i=1}^{t} i = (t^2+t)/2$. Thus $t^2 < 2d$; so $t < (2d)^{1/2}$. ■

Let P be an arbitrary r-variate, r > 0, polynomial. The (r-1)-variate content of P, written cont(r-1,P), is cont(P). For $0 \leq j < r-1$, the j-variate content of P, written cont(j,P), is cont(cont(j+1,P)). Thus, for $1 \leq j < r$ cont(j,P) is a j-variate polynomial and cont(0,P) is a constant. The r-variate primitive part of P, written pp(r,P), is pp(P). For $1 \leq j < r$, the j-variate primitive part of P, written pp(j,P), is the ample associate of pp(cont(j+1,P)). Observe that pp(j,P) is a j-variate polynomial. Furthermore, $P \sim \text{cont}(0,P)\prod_{j=1}^{r} \text{pp}(j,P)$. The primitive squarefree factorization of P is the list $L = (L_1,\ldots L_r)$ where L_j is the squarefree factorization of pp(j,P). Thus $B_j = \{b\in L_j: b \text{ is non-constant}\}$ is a basis for pp(j,P) and $B = \bigcup_{j=1}^{r} B_j$ is a basis for P. The algorithm GPPSQF specified below obtains the primitive squarefree factorization of a Gaussian polynomial. Its description may be found in [EPS75].

L = GPPSQF(P)

Input: P, an r-variate, r > 0, Gaussian polynomial.
Output: $L = (L_1,\ldots,L_r)$, the primitive squarefree factorization of P.

Now let R be an r-variate, r > 0, rational function in $U[x_1,\ldots,x_r]$. Let B_1 be a basis for {num(R)} and B_2 be a basis for {den(R)}. Since num(R) and den(R) are relatively prime, each polynomial in B_1 is relatively prime to each polynomial in B_2. Therefore, $B_1 \cup B_2$ is a basis for {num(R), den(R)} as well as being a basis for {R}. Thus, a basis for {R} can be obtained from the primitive squarefree factorizations of num(R) and den(R) by omitting the constant polynomials in the output. Furthermore, by keeping the 1-variate, 2-variate,..., and r-variate polynomials obtained in the primitive squarefree factorizations of num(R) and den(R) separate, it is possible to find bases for {pp(j,num(R))/pp(j,den(R))}, $1 \leq j \leq r$. The algorithm GRPSQF uses GPPSQF to produce the lists $(B_1,\ldots,B_r)$ and $(E_1,\ldots,E_r)$ where for $1 \leq j \leq r$ B_j is a basis for {pp(j,num(R))/pp(j,den(R))} and E_j is the corresponding multiplicity sequence.

GRPSQF(R, B, E)

Input: R, an r-variate, r > 0, Gaussian rational function.
Output: $B = (B_1,\ldots,B_r)$ and $E = (E_1,\ldots E_r)$ where for $1 \leq j \leq r$ B_j is a basis for {pp(j,num(R))/pp(j,den(R))} and E_j is the corresponding multiplicity sequence.

Description
(1) [Initialize k and obtain the squarefree factorizations of the numerator and

denominator of R.] $k \leftarrow 0$; $(N_1,\ldots,N_r) \leftarrow$ GPPSQF(num(R)); $(D_1,D_2,\ldots,D_r) \leftarrow$ GPPSQF(den(R)).

(2) [Update k, s and t.] $k \leftarrow k+1$; $s \leftarrow$ the number of elements in N_k; $t \leftarrow$ the number of elements in D_k.

(3) [$N_k = (N_{k,1},\ldots,N_{k,s})$; $D_k = (D_{k,1},\ldots,D_{k,t})$; form B_k and E_k.] For i=1 to s do,(if $N_{k,i} \neq 1$, ($j \leftarrow j+1$; $B_{k,j} \leftarrow N_{k,i}$; $E_{k,j} \leftarrow i$)); for i=1 to t do, (if $D_{k,i} \neq 1$, ($j \leftarrow j+1$; $B_{k,i} \leftarrow D_{k,i}$; $E_{k,j} \leftarrow -i$)); $B_k \leftarrow (B_{k,1},\ldots,B_{k,j})$; $E_k \leftarrow (E_{k,1},\ldots,E_{k,j})$.

(4) [Check whether k=r.] If $k<r$, go to (2); $B \leftarrow (B_1,\ldots,B_k)$; $E \leftarrow (E_1,\ldots E_k)$; return.

GRPSQF can be used in obtaining a basis and multiplicity matrix for a single Gaussian rational function, but a procedure for building two bases and multiplicity matrices into a single basis and multiplicity matrix is now needed. The next algorithm uses a procedure suggested by Collins in [COL74] to build a single basis from two bases and expands on his idea so that the multiplicity matrix corresponding to the new basis is created as the basis is created.

Let R and S be sets of rational functions and let A be a basis for R and B be a basis for S. The only way that $A \cup B$ can fail to be a basis for $R \cup S$ is that some A_i in A and B_k in B are not relatively prime. Thus, the main task of the next algorithm BASISJ is to find $\gcd(A_i,B_k)$ for each A_i in A and B_k in B. Whenever $\gcd(A_i,B_k) \sim 1$, A_i and B_k are left alone. Otherwise, $G = \gcd(A_i,B_k)$ is put in the new basis, A_i is replaced by A_i/G, and B_k is replaced by B_k/G. Since G is a factor of B_k, it is relatively prime to each other element of B. Because B_k is squarefree, $\gcd(G,B_k/G) \sim 1$. Thus G is relatively prime to each element of B once B_k is replaced by B_k/G. Similarly, G is relatively prime to each element of A after A_i is replaced by A_i/G. Thus, after each $\gcd(A_i,B_k)$ is computed, and after the necessary replacements of A_i by A_i/G and B_k by B_k/G are made, the set of ample associates of the A_i, B_k and G which are non-constant will be a basis for $R \cup S$.

While forming the new basis, BASISJ forms its multiplicity matrix as follows. Suppose $(f_{i1},f_{i2},\ldots,f_{iq})$ is the occurrence sequence of A_i relative to $R = \{R_1,\ldots,R_q\}$ and $(e_{k1},e_{k2},\ldots,e_{km})$ is the occurrence sequence of B_k relative to $S = \{S_1,\ldots,S_m\}$. Let $G = \gcd(A_i,B_k)$. If $G \sim 1$, G is not in the basis. Otherwise, $(f_{i1},f_{i2},\ldots,f_{iq}, e_{k1},e_{k2},\ldots,e_{km})$ is the occurrence sequence of G relative to $\{R_1,\ldots,R_q,S_1,\ldots,S_m\}$. After each $\gcd(A_i,B_k)$ has been computed and all necessary replacements made, each A_i is relatively prime to each B_k. Thus no A_i is a factor of any S_j. So, if $(f_{i1},f_{i2},\ldots,f_{iq})$ was the occurrence sequence of A_i relative to R at the start of the algorithm BASISJ, $(f_{i1},f_{i2},\ldots,f_{iq}, \underbrace{0,\ldots,0}_{m \text{ zeros}})$ is the occurrence sequence of A_i relative to $R \cup S$. Similarly, if $(e_{k1},e_{k2},\ldots,e_{km})$ is the occurrence sequence of B_k at the start of the algorithm, $(\underbrace{0,\ldots,0}_{q \text{ zeros}}, e_{k1},\ldots,e_{km})$ is the occurrence sequence of B_k relative to $R \cup S$. The following algorithm description gives the details of the algorithm BASISJ.

BASISJ(B, E, A, F, C, M)

Input: $B = (B_1,\ldots,B_n)$, a basis for $S = \{S_1,\ldots,S_m\}$ where each B_k in B is a primitive r-variate Gaussian polynomial;

$E = \begin{pmatrix} e_{11} & \cdots & e_{1,m} \\ \vdots & & \vdots \\ e_{n1} & \cdots & e_{n,m} \end{pmatrix}$, the multiplicity matrix for B with respect to S;

$A = (A_1,\ldots,A_t)$, a basis for $R = \{R_1,\ldots,R_q\}$ where each A_i in A is a primitive r-variate Gaussian polynomial;

$F = \begin{pmatrix} f_{11} & \cdots & f_{1,q} \\ \vdots & & \vdots \\ f_{t1} & \cdots & f_{t,q} \end{pmatrix}$, the multiplicity matrix for A with respect to R.

Output: C, a basis for $R \cup S$ consisting of primitive r-variate Gaussian polynomials; M, the multiplicity matrix for C with respect to $R \cup S$.

Description

(1) [Initialize.] $j \leftarrow 0$; For k=1 to n, $B_k^* \leftarrow B_k$.

(2) [Compute $G = \gcd(A_i^*, B_k^*)$; determine whether G and A_{*i}^* belong in C; form their occurrence sequence if they do.] For i=1 to t, ($A_i^* \leftarrow A_i$; For k=1 to n, ($G \leftarrow \gcd(A_i^*, B_k^*)$; if $G \neq 1$, ($j \leftarrow j+1$, $C_j \leftarrow G$; $M_j \leftarrow (f_{i1},\ldots,f_{iq}, e_{k1},\ldots,e_{km})$; $A_i^* \leftarrow A_i^*/G$; $B_k^* \leftarrow B_k^*/G$)); if $\deg(A_i^*) \neq 0$, ($j \leftarrow j+1$; $C_j \leftarrow FQV(A_i^*)$; $M_j \leftarrow (f_{i1},\ldots,f_{iq}, 0,\ldots,0)$).

(3) [Find the non-constant B_k^*'s and form the corresponding occurrence sequences.] For k=1 to n, (if $\deg(B_k^*) \neq 0$, ($j \leftarrow j+1$; $C_j \leftarrow FQV(B_k^*)$; $M_j \leftarrow (0,\ldots,0,e_{k1},\ldots, e_{km})$).

(4) [Finish up and return.] $C \leftarrow (C_1,\ldots,C_j)$; $M \leftarrow (M_1,\ldots M_j)$; return.

Let $R = \{R_1,\ldots,R_m\}$ be a set of r-variate rational functions with coefficients in the ufd U. Suppose $\text{num}(R_i)$ and $\text{den}(R_i)$ are primitive polynomials for each i, $1 \leq i \leq m$. Suppose A_i is a basis for $\{R_i\}$ with corresponding multiplicity matrix F_i, and B_i is a basis for $\{R_1,\ldots,R_{i-1}\}$ with corresponding multiplicity matrix E_i. By applying BASISJ successively with inputs B_i, E_i, A_i and F_i, $2 \leq i \leq m$, a basis C for R and the corresponding multiplicity matrix M can be obtained.

In general, it is not known that each $\text{num}(R_i)$ and $\text{den}(R_i)$ is primitive. However, each $\text{num}(R_i)$ may be written $\text{num}(R_i) = p_i \prod_{j=1}^{r} P_{ij}$ where p_i is in U and $P_{ij} = pp(j,\text{num}(R_i))$. Also, $\text{den}(R_i) = q_i \prod_{j=1}^{r} Q_{ij}$ with q_i in U and $Q_{ij} = pp(j,\text{den}(R_i))$. Then, for each j = 1,2, ...,r, consider the set of rational functions $S_j = \{P_{1j}/Q_{1j}, P_{2j}/Q_{2j},\ldots, P_{mj}/Q_{mj}\}$. Certainly each pair P_{ij}, Q_{ij} is a pair of ample, primitive relatively prime j-variate polynomials. Thus, the inductive procedure suggested above can be used to form a basis B_j of j-variate polynomials and the corresponding multiplicity matrix, M_j, for S_j. Since each P_{ij} and Q_{ij} is primitive, each element of B_j must also be primitive. Therefore, whenever b_i is in B_i, b_j is in B_j, and $i \neq j$, b_i and b_j are relatively prime. $\bigcup_{j=1}^{r} B_j$ will then be a basis for $\bigcup_{j=1}^{r} S_j$.

The multiplicity matrix M is obtained from the M_j as follows. If M_j has r_j rows, then M will have $\sum_{j=1}^{r} r_j$ rows. The first r_1 rows will be those of M_1. Then, the next r_2 rows will be the rows of M_2 and so on. The algorithm BASIS is used to form the B_j and the corresponding M_j.

BASIS(R, B, M)

Input: $R = \{R_1,\ldots,R_m\}$, a set of $m \geq 1$ r-variate rational functions.

Output: $B = (B_1,B_2,\ldots,B_r)$ and $M = (M_1,M_2,\ldots,M_r)$ where for $1 \leq j \leq r$ B_j is a basis for $S_j = \{pp(j,\text{num}(R_i))/pp(j,\text{den}(R_i)): 1 \leq i \leq m\}$ and M_j is the corresponding multiplicity matrix.

Description

(1) [Apply GRPSQF to R_m thus initializing B to $(B_1,\ldots,B_m)$ and M to $(M_1,\ldots,M_j)$ where B_j is a basis for $pp(j,\text{num}(R_m))/pp(j,\text{den}(R_m))$ and M_j is the corresponding multiplicity matrix.] Call GRPSQF(R_m, B, M).

(2) [Apply GRPSQF to R_k obtaining $A = (A_1,\ldots,A_r)$ and $E = (E_1,\ldots,E_r)$ where A_j is a basis for $pp(j,\text{num}(R_k)/pp(j,\text{den}(R_k))$ and E_j is the corresponding multiplicity matrix. Merge B_j and A_j and obtain the corresponding multiplicity matrix.] For k = m-1,..., 2,1, (Call GRPSQF(R_k, A, E); for j = 1 to r, call BASISJ(B_j, M_j, A_j, E_j, B_j, M_j)).

(3) [Finish up and return.] $B \leftarrow (B_1,\ldots,B_r)$; $M \leftarrow (M_1,\ldots,M_r)$; return.

Using BASIS it is an easy matter to obtain a matrix which represents the system of homogeneous linear equations (*) mentioned in Theorem 1. Since efficient algorithms for solving such systems of equations are known (see [McC73]), assume an algorithm with the following specifications is available.

S = LINEQS(M)

Input: $M = (M_1,\ldots,M_r)$, $r \geq 1$, where each M_j, $1 \leq j \leq r$, is a t_j x m matrix with integer entries.

Output: S; when the system of equations $M_jX = 0$, $1 \leq j \leq r$ has only the trivial solution, S = 0; otherwise S is an m x n matrix with integer entries whose columns are a maximal linearly independent set of solutions to the system $M_jX = 0$, $1 \leq j \leq r$.

It is now an easy matter to determine whether a set, $R = \{R_1,\ldots,R_m\}$, of r-variate, $r \geq 1$, rational functions are pseudo-multiplicatively independent. For, when BASIS is applied to R, one of the outputs is $M = (M_1,\ldots,M_r)$. By Theorem 1, if $(n_1,\ldots n_m)$ is any integer solution to $M_jX = 0$, $1 \leq j \leq r$, $\prod_{i=1}^{m} R_i^{n_i}$ will be a constant. This leads to the algorithm PMULTI.

S = PMULTI(R)

Input: $R = \{R_1,\ldots,R_m\}$, a set of r-variate, $r \geq 1$, non-constant Gaussian rational functions.

Output: S; if R is pseudo-multiplicatively independent, S = 0; otherwise, S = $(n_1,\ldots,n_m)$ is a list of integers such that $\prod_{i=1}^{m} R_i^{n_i}$ is a constant.

Description

(1) [Determine M and S.] Call BASIS(R, B, M); S $\leftarrow$ LINEQS(M); if S $\neq$ 0, S $\leftarrow$ first column of S; return.

4. Theoretical Computing Time Analysis

In this section the computing times of the algorithms described in the previous section are discussed. The discussion here will be far from exhaustive. Those who desire a fuller discussion are referred to pages 123-158 of [EPS75] where detailed analyses of the SAC-1 implementation of algorithms similar to those described here are presented.

Let A be an algorithm with n inputs. Then the time required to execute A with valid inputs $I_1,\ldots,I_n$, denoted $t_A(I_1,\ldots,I_n)$, is called the computing time of A. The function t_A is called the computing time function for algorithm A. If f and g are two non-negative real valued functions on some domain S, f is dominated by g, written $f \precsim g$, in case there is a constant $c > 0$ such that $f(x) \leq cg(x)$ for each x in S.

The length of a rational integer a, denoted L(a), is $\log_2|a|+1$. The length of an integer polynomial $P(x_1,\ldots,x_r) \neq 0$, denoted $L(P(x_1,\ldots,x_r))$, is the maximum of the lengths of its integer coefficients. Let L(0) = 1. Let A,B be rational integers (or integer polynomials). The length of the Gaussian integer (or polynomial) A + Bi, denoted L(A + Bi), is $\max\{L(A), L(B)\}$. Let G [r,d,a] be the collection of all r-variate Gaussian polynomials, P, with degree $\leq$ d in each variable and $L(P) \leq a$. G(r,d,a) will denote the set of all Gaussian rational functions, R, with num(R) and den(R) in G [r,d,a].

It is assumed that modular algorithms are used by PMULTI to calculate greatest common divisors and in the linear equation solver. These modular algorithms have much lower computing time bounds than the classical algorithms if one assumes that few modular homomorphisms or substitution homomorphisms are unsuccessful and, in the case of the linear equation solver, if what McClellan [McC73] calls premature equality test success does not occur. As pointed out in [McC73], any of these occurrences are quite improbable, so assume none of these events occur. Also, assume, as in the analysis in [CAR75], that if P is a factor of the Gaussian polynomial Q, $L(P) \leq L(Q)$. With these assumptions, it can be shown (see [CAR75]) that $t_{gcd}(A,B) \precsim a^2(2d+1)^{r+1}$ for A,B in G [r,d,a] and that the time required to solve a system of t equations in m unknowns $\precsim tm^3L(ma)^2$ where a is the maximum of the absolute values of all coefficients in the equations (see [McC73]). Using the above bound for the computing time of the gcd algorithm, the analysis of GPPSQF in [EPS75] can be modified to show that $t_{GPPSQF}(P)$

$\simeq a^2(2d+1)^{r+2}$ for P in G [r,d,a]. Now the following theorem can be proved.

Theorem GRPSQF. Let R be a non-constant function in G(r,d,a). Then $t_{GRPSQF}(R) \simeq a^2(2d+1)^{r+2}$.

Proof. GRPSQF works by invoking GPPSQF twice, once to factor num(R) and once to factor den(R), and then scanning the resulting lists to find the factors $\neq 1$ and form a list of their multiplicities. Clearly, the time required to create the lists of factors dominates the time required to scan the factors and create the multiplicity sequence. Thus, the $t_{GRPSQF}(R) \simeq t_{GPPSQF}(num(R)) + t_{GPPSQF}(den(R)) \simeq a^2(2d+1)^{r+2}$. ■

The modular gcd algorithm for Gaussian polynomials can be used to calculate not only $G = gcd(A,B)$ but $A^* = A/G$ and $B^* = B/G$. This gcd and cofactor algorithm also has a computing time function $\simeq a^2(2d+1)^{r+2}$ when applied to inputs A and B in G [r,d,a]. The analysis of BASISJ below will assume that this gcd and cofactor algorithm is used to calculate the cofactors in step(2) of BASISJ rather than calculating the cofactors $A^* = A/G$ and $B^* = B/G$ by first calculating $G = gcd(A,B)$ and then using a classical division algorithm. This assumption leads to the following theorem.

Theorem BASISJ. Let B be a basis for R with E the corresponding multiplicity matrix and A be a basis for S with F the corresponding multiplicity matrix. Suppose B is a list of $n>0$ elements in G [r,d,a], A is a list of $t>0$ elements in G [r,d,a], and there are m elements in $R \cup S$. Then $t_{BASISJ}(B, E, A, F) \simeq mnt + nta^2(2d+1)^{r+1}$.

Proof. Step(1) takes time $\simeq n$. Step(2) requires calculating $G = gcd(A_i^*, B_k^*)$ for each $1 \leq i \leq t$, $1 \leq k \leq n$ as well as the cofactors A_i^*/G and B_k^*/G whenever $G \neq 1$. Each A_i^* (or B_k^*) is either a factor of an element of A (or B). Thus A_i^* and B_k^* are in G [r,d,a]. So the time required to find each $gcd(A_i^*, B_k^*)$ as well as cofactors A_i^*/G and B_k^*/G $\simeq a^2(2d+1)^{r+1}$. The time required to find the first quadrant value of any cofactor is dominated by the time required to calculate the cofactor. Thus, the time required to determine all of the C_j found in step(2) $\simeq nta^2(2d+1)^{r+1}$. The time required to form any list M_j is dominated by the length of that list, which is m the number of elements in $R \cup S$. Thus, the total time spent in executing step(2) $\simeq mnt + nta^2(2d+1)^{r+1}$

The total time spent in executing step(3) is dominated by the time required to find the first quadrant value of the B_k^* and the time required to form the M_j. Thus, step(3) takes time $\simeq n(m + a^2(2d+1)^{r+1})$.

The time required to execute step(4) is dominated by the time required to form C and M, and thus is dominated by the length of C (which equals the length of M). Since each element of C is either a $gcd(A_i^*, B_k^*)$ or an A_i^* or a B_k^*, there are at most $nt+t+n \simeq nt$ elements in C. Thus nt dominates the time required for step(4). The theorem now follows easily by adding the bounds on the computing times of each step. ■

Theorem BASISJ leads to the following theorem giving a bound for t_{BASIS}.

Theorem BASIS. Let $R = \{R_1, \ldots, R_m\}$ be a set of $m > 0$ non-constant functions G(r,d,a). Then $t_{BASIS}(R) \simeq a^2m^3d^{3/2}(2d+1)^{r+1}$.

Proof. The time required by step(1) $\simeq t_{GRPSQF}(R_1) \simeq a^2(2d+1)^{r+2}$ since R_1 is in G(r,d,a).

For each of $R_2, \ldots, R_m$, step(2) calls GRPSQF once, taking time $\simeq a^2(2d+1)^{r+2}$. Also, for each R_i, $2 \leq i \leq m$, BASISJ is called r times, once to update each B_j, $1 \leq j \leq r$. When BASISJ is called to update B_j for the k-th time, the inputs are A_j, a basis for $\{pp(j,num(R_k))/pp(j,den(R_k))\}$, E_j, the corresponding multiplicity matrix, B_j, a basis for $S_k = \{pp(j,num(R_i))/pp(j,den(R_i)) : 1 \leq i \leq k\}$, and M_j, the corresponding multiplicity matrix. S_k is a set of k-1 elements in G(j,d,a) each with a primitive numerator and denominator. Thus, B_j has at most 2(k-1)d elements. It follows from Theorem 2 that A_j has at most $2(2d)^{1/2}$ elements. Thus, Theorem BASISJ implies that the time required to update B_j and M_j for the k-th time is $\simeq (k-1)d^{3/2}(k+a^2(2d+1)^{j+1})$. The time required for

all calls to BASISJ $\simeq \sum_{j=1}^{r} \sum_{k=2}^{m} (k-1)d^{3/2}(k+a^2(2d+1)^{j+1}) \simeq rm^3d^{3/2} + m^2d^{3/2}a^2(2d+1)^{r+1}$ $\simeq a^2m^3d^{3/2}(2d+1)^{r+1}$. It follows that step(2) takes time $\simeq a^2m^3d^{3/2}(2d+1)^{r+1}$. It is clear that step(3) takes time $\simeq r$. The theorem now follows easily by adding the time bounds on steps (1), (2), and (3). ■

Combining the bound on $t_{BASIS}(R)$ with McClellan's bound on the time required by the linear equation solver yields the following theorem.

Theorem PMULTI. Let $R = \{R_1,\ldots,R_m\}$ be a set of $m > 0$ non-constant functions in $G(r,d,a)$. $t_{PMULTI}(R) \simeq m^4rdL(md)^2 + a^2m^2d^{3/2}(2d+1)^{r+1}$.

Proof. The time required for the call to BASIS $\simeq a^2m^3d^{3/2}(2d+1)^{r+1}$. This call produces M, a list of r matrices. Each of these matrices represents a set of equations in m unknowns; each equation represents a different non-constant factor of $num(R_i)$ or $den(R_i)$ for some i. Since each R_i is in $G(r,d,a)$, each R_i can have at most 2rd distinct non-constant factors. Therefore, M represents a system of at most 2rdm homogeneous equations in m unknowns. The coefficients in these equations correspond to the multiplicities of non-constant factors of $num(R_i)$ and $den(R_i)$, $1 \leq i \leq m$. Since each R_i is in $G(r,d,a)$, each coefficient $\leq d$. Thus, it follows from McClellan's bound on the time required by the linear equation solver that $\simeq rdm^4L(md)^2$. The theorem now follows easily. ■

5. Conclusions

Various modifications to the algorithms described in section 3 are possible and may be desirable for various purposes. For example, only minor modifications to BASISJ would be required in order to apply the algorithms presented to rational functions in $U(x_1,\ldots,x_r)$, where U is any ufd with characteristic 0, in which gcd's, quotients, and formal derivatives can be effectively computed. The formal derivatives are needed in the squarefree factorizations.

A slight reorganization can speed up the algorithm in many cases where the rational functions are pmi. For, if one of the matrices M_j produced by BASIS, $M_jX = 0$ has only the trivial solution, the system $M_jX = 0$, $1 \leq j \leq r$, can have only the trivial solution. Reorganizing the algorithms so that the basis B_k and the matrix M_k is not computed unless a non-trivial solution to the system $M_jX = 0$ for $k \leq j \leq r$ has been found can speed the algorithm in cases where the rational functions are pmi. Of course, if the rational functions are not pmi, this approach adds to the overhead by requiring the solution of several systems of linear equations unnecessarily. A linear equation solver which makes use of a general solution to the system $M_jX = 0$, $k \leq j \leq r$, in solving $M_jX = 0$, $k \leq j \leq r$ would greatly enhance this approach.

In addition to being somewhat faster when the functions turn out to be pmi, the present organization of the algorithms has the advantage of producing a basis. Basis computation has some inherent interest of its own apart from its usefulness in the present context. Collins [COL74], for example, found it useful in quantifier elimination problems.

References

[CAR75] B. F. Caviness, M. Rothstein, A Modular Greatest Common Divisor Algorithm For Gaussian Polynomials, University of Wisconsin Computer Sciences Dept. Tech. Report #245, March, 1975.

[COL74] G. E. Collins, Quantifier Elimination in Real Closed Fields by Cylindrical Decomposition, ACM SIGSAM Bulletin, Vol. 8, No. 3, August 1974, pp. 80-90.

[EPS75] H. I. Epstein, Algorithms for Elementary Transcendental Function Arithmetic, Ph.D. Thesis, University of Wisconsin, Madison, Wisconsin, April, 1975, available through University Microfilms, Ann Arbor, Michigan.

[FAT72] R. J. Fateman, Essays in Algebraic Simplification, Project MAC TR-95, Massachusetts Institute of Technology, Cambridge, Mass., April, 1972.

[JOH71] S. C. Johnson, On the Problem of Recognizing Zero, JACM, Vol. 18, October, 1971, 1971, pp. 559-565.

[McC73] M. T. McClellan, The Exact Solution of Systems of Linear Equations with Polynomial Coefficients, JACM, October, 1973, Vol. 20, No. 4, pp. 563-588.

On the Sign of a Real Algebraic Number

Siegfried Rump
Universität Kaiserslautern

March 1976

Abstract
In an ordered algebraic extension field of the rationals algorithms for sign determinations are studied. Two new algorithms are analyzed in detail and shown to be asymptotically and in practice faster than previous algorithms.

1. Introduction

Let Z denote the ring of integers and Q the field of rationals. In a pure algebraic sense an algebraic number α over Q may be represented by its minimal monic polynomial $\psi \in Q[x]$, $\psi(\alpha) = 0$. It is possible [1,2] to perform constructively the four rational operations in $Q(\alpha)$ using only ψ to characterize α and the same holds for operations composed of the four rational operations like greatest common divisor calculation and factorization. There are, however, circumstances where one has to distinguish between an algebraic number α and its conjugates belonging to the same ψ. Very important is the case that the field is ordered like Q and the order relation has to be extended to $Q(\alpha)$. Obviously, ψ is not sufficient to characterize α anymore. Zassenhaus proposed and realized in his real root calculus [3] an indexing from left to right of the real algebraic numbers of a minimal ψ and denotes the roots by $\alpha(\psi,1)$, $\alpha(\psi,2),\ldots,\alpha(\psi,r)$ if ψ has r distinct real roots. The sign determination of an element $\beta \in Q(\alpha)$ as the base for the order relation depends in general on all real roots of φ, $\varphi(\alpha) = \beta$, and ψ, We call such an algorithm based on all real roots of φ a global algorithm for sign determination.

In section 2 we state the problem more formally and discuss global algorithms. Using another indexing method which was introduced by Heindel [4] based on isolating intervals we describe and analyze in section 3 and 4 two new algorithms which are not global in the stated sense. In section 5 we give empirical comparisons and discuss finally applications of the given algorithms for real root isolation of real algebraic polynomials.

2. Global Algorithms

Some of the restrictions we use in the formal statement of the problem will be lifted later for reasons of computational convenience. If ψ is the minimal polynomial over Q[x] such that $\psi(\alpha) = 0$ then every element $\beta \in Q(\alpha)$ can be represented by a polynomial $B \in Q[x]$ of degree $n_B \leq n_\psi - 1$. Let us denote by Ω an interval with rational endpoints r and s such that $\alpha \in \Omega$ but no other real root of ψ is contained in Ω. Ω is called an isolating interval for α with respect to ψ. Since ψ is minimal, $\gcd(\psi,B) = 1$ if $B \neq 0$. Therefore $\beta = 0$ is represented by $B = 0$, the null polynomial, and the problem of sign determination is reduced to the non-zero sign of β, hence to sign $(B(\alpha)) \neq 0$ (Figure 1).

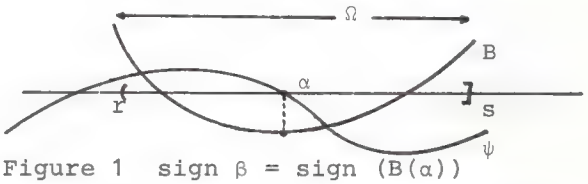


Figure 1 sign β = sign $(B(\alpha))$

For the indexing method of Zassenhaus a solution using Sturm sequences was given by Kempfert [11], and for Collins' method of isolating intervals by Rubald [2]. Rubald's algorithm counts the number of real roots of B in Ω and bisects Ω with respect to α until Ω does not contain a root of B anymore. Then sign (β) = sign $(B(r))$ = sign $(B(s))$. The Sturm sequence is generated once and evaluated repeatedly for each new Ω. We call this algorithm the S-algorithm (for Sturm sequence) which is available in the SAC-1 system [2] for computer algebra.

Recently two algorithms were discovered which can be used as alternatives to Sturm's method. The first one [5] isolates the real roots of a polynomial using the sequence of derivatives, the second one [6] is based on an idea of Uspensky and uses polynomial transformations and Descarte's sign rule. Both algorithms can be easily adapted for an arbitrary interval like Ω. Both algorithms are in general superior to Sturm's algorithm and it can be expected that sign algorithms based on them are also superior to the S-algorithm.

The global algorithms can be described as follows

s ← ASIGNG(α,B)

[Algebraic sign, global algorithm. $\alpha=(\Omega,\psi)$, $\beta=B(\alpha)$. s=sign(β)]
(1) [β=0?] if B=0 then set s=0 and stop.
(2) [Isolating intervals] Let $I_1,\ldots,I_{r_B}$ be the isolating intervals of B, using one of the three mentioned algorithms.
(3) [Bisect] while there are intervals $I_n,\ldots,I_m$ such that $I_i\cap\Omega\neq\emptyset$, $n\leq i\leq m$, do {bisect Ω with respect to ψ and $I_n,\ldots,I_m$ with respect to B}.
(4) [Sign] Set s=sign(B(r)), where Ω=(1,r) and stop.

3. The Interval Algorithm and the Derivative Algorithm

If $B\neq 0$ then there exists a lower bound b on (B(α)) which we will give in the next section. Also let us define the interval Y=B(Ω) by evaluating the polynomial B using interval addition, subtraction and multiplication. A sequence of intervals $\Omega=\Omega_1,\Omega_2,\ldots,\Omega_m$ bisected with respect to ψ such that $\alpha\in\Omega_i$, results by $Y_i=B(\Omega_i)$ in a sequence of intervals $Y_1,\ldots,Y_m$, with $B(\alpha)\in Y_i$ and $\Omega_{i+1}\subset\Omega_i$, $Y_{i+1}\subset Y_i$ for $i=1,\ldots,m-1$. Since the width of the intervals in both sequences decreases monotonically m can be choosen such that the width of Y_m is less then the bound b. Hence $0\notin Y_m=(l,r)$ and sign (B(α))=sign(r). This leads to the following algorithm [7]

s ← ASIGNI(α,B)

[Algebraic sign using interval arithmetic. $\alpha=(\psi,\Omega)$, $\beta=B(\alpha)$. **s=sign(β)**]
(1)[β=0?] if B=0 then set s=0 and stop.
(2)[Bisect] Y←B(Ω); while $0\in Y$ do {bisect Ω with respect to ψ and set Y←B(Ω) using interval arithmetic}.
(3)[Sign] Let Y=(l,r); Set S=sign(l) and stop

Although ASIGNI turns out to be asymptotically and in practice faster than the S-algorithm there is still another method empirically more attractive.

The algorithm is based on the following observation. If the derivative B'(x)=dB/dx has no real root in Ω , then B has by Rolle's Theorem at most one real root in Ω , which is of multiplicity 1 if it exists. Then the sign of B is different on the endpoints of Ω. (The case that Ω is of width 0 means that $\alpha\in Q$ ans may be trivially excluded). Bisecting Ω with respect to ψ until the signs of B at the endpoints agree results in an interval Ω on which B is of the same sign. It remains to enforce the hypothesis. This can easily be accomplished inductively because the n_B-th derivative of B is a non-zero constant.

s ← ASIGND(α,B)

[Algebraic sign using derivatives. $\alpha=(\psi,\Omega)$, $\beta=B(\alpha)$. s=sign(β)]
(1) [β=0?]if B=0 then set s=0 and stop.
(2) [Derivative sequence] Let $B^{(0)}$ $B^{(1)},\ldots,B^{(n_B)}$ be the derivative sequence of B.
(3) [Induction] for $k=n_B,n_B-1,\ldots,0$ do with Ω=(1,r) {while sign $(B^{(k)}(1))\neq$ sign $(B^{(k)}(r))$ bisect Ω with respect to ψ}.
(4) [Sign] Set s = sign(B(r)) and stop.

The algorithm as stated is only correct for minimal ψ. This restriction can easily be lifted.

4. Theoretical Analysis

The assumptions made in the actual implementation are the following. ψ is not assumed to be irreducible since there exists no complete factorization algorithm with a polynomial bound on the computing **time.** Instead ψ is assumed to be square-**free** and primitive since $\psi\in Z[x]$. B is assumed to be relatively prime to ψ, otherwise B=0. B is split into a rational number r_B and a primitive polynomial $\bar{B}\in Z[x]$. Since $r_B>0$ only $\bar{B}$ is of interest in the algorithm.

We denote by L(a) the length of the integer a, by L(r)=max{L(a),L(b)} the length of the rational r=a/b reduced to lowest terms and by L(I)=max{L(l),L(r)} the length of the interval Ω=(l,r) with rational endpoints l and r. The width W(Ω)=|r-l| is distinguished from L(Ω). Finally, the integral polynomial A is characterized by its degree n_A and its sum norm $d_A=\sum_{0\leq i\leq n_A}|a_i|$ where the a_i are the coefficients of A.

Let the interval (-a,a] contain all real roots of $B\in Z[x]$. If a is an integer all intervals occuring in the algorithms generated by bisection of such an initial interval have binary rational endpoints,i.e. the denominator is a power of 2. Binary rational arithmetic is much cheaper than rational arithmetic. For the bisection of the intervals Collins' [5] method of minimizing the length of the numerators and denominators is employed.

The analysis of algorithm ASIGNG gives for step 2

$$t_2 \leq n_B^{10} + n_B^7\, L(d_B)^3$$

for Sturm and derivate sequence [4,5]

and $t_3 \preceq t_2$ by the argument in [5] for algorithm R if $L(d)=\max\{L(d_B),L(d_\psi)\}$. Hence, for $n_\psi = n > n_B$,

$$t_{ASIGNG} \preceq n^{10} + n^7 L(d)^3.$$

In practice, one does not have to generate all isolating intervals in step 2 in the S-algorithm, if one bisects Ω until it contains only the nearest root of B to α. Then only one applies step 3. With this improvement Rubald's S-algorithm has the time

$$t_S \preceq n^9 + n^6 L(d)^3.$$

The use of the stronger result of Mignotte [12], [13] does not affect this time.

If in step 2 the modified Uspensky algorithm is used we have

$$t_{ASING} \preceq n^8 + n^6 L(d)^2 + n^5 L(d)^3.$$

where the second term is due to step 3.

The analysis of the interval algorithm ASIGNI is based on Theorem 5 of [5] which gives the lower bound

$$|B(\alpha)| > \frac{1}{2}\,(d_B+1)^{-n_\psi} d_\psi^{-n_B}$$

and on the relation between the width of $Y = B(\Omega)$ and of Ω

$$W(Y) < n_B \omega^{n_B-1} d_B \cdot W(\Omega)$$

where $\omega = \max\{|r|,|1|\} + W(\Omega) + 1, \Omega=(1,r]$.

Theorem Let $\alpha = (\psi,\Omega), \beta = B(\alpha) \neq 0$,
$n = n_\psi > n_B$, $d = \max\{d_\psi,d_B\}$
Then $t\,(ASIGNI,\alpha,B) \preceq n^5 L(d)^3$ and
$t\,(ASIGND,\alpha,B) \preceq n^8 + n^5 L(d)^3$.

For algorithm ASIGND it is not obvious that no root of the derivatives coincides with α. This cannot happen if ψ is minimal; otherwise by calculating the $\gcd(\psi,B^{(k)})$ ψ can always be made relatively prime to $B^{(k)}$, $0 < k < n_B$. Even without assuming a minimal ψ we get the stated result for ASIGND. Collins and Horowitz [14] have shown, that a square free polynomial with degree n and sumnorm $\underline{d}_1$ have a minimum root separation λ with $L(\lambda^{-1}) \preceq n \cdot L(d)$. One can show, that this result still holds for the distinct roots of a polynomial with possibly multiple roots. It is conjectured, that the minimum real root separation of a polynomial as given by the formula is also a minimum real root separation for all the derivatives. This has not yet been proved and in the derivation of the computing time of ASIGND a λ, with $L(\lambda^{-1}) \preceq n^2+nL(d)$, is used as a minimum root separation for all derivatives contributing to the term n^8 in the result. Detailed proofs will be contained in [8].

5. Empirical Analysis

For Rubald's S-algorithm, ASIGNI and ASIGND we give in the following tables in columns S, I and D times in seconds of a TR 440 computer using SAC-1 [9]. For each algorithm we give also under B the number of bisections. Since we took the average of three randomly generated polynomials for each entry B is a fraction. With the exception that Ω is initially sufficient small that no bisections are needed at all, algorithm ASIGND is clearly superior to ASIGNI and both are faster than the Sturm sequence algorithm. The efficient modification of Uspensky's algorithm became only known while this note was written. Therefore, no tests could be included. b denotes the length of the coefficients in bits, n the degree of the polynomials, ψ has always 44 bit (1 word on TR 440) coefficients and degree n + 1.

Table 1 - Random Polynomials, b = 44

n	S	B	I	B	D	B
5	2.8	10.3	2.6	10.6	1.0	13.6
10	21.2	12.3	6.1	12.0	1.5	12
10*)	9.7	0	0.6	0	1.09	2
15	46.0	3	7.5	11.7	3.3	15.3
20	-	-	13.4	15.3	4.4	15.3
20*)	140	1.3	4.63	3.1	4.60	3.3

*) Ω three orders of magnitude smaller

Table 2 - Random Polynomials, n = 10

b	S	B	I	B	D	B
44	12.8	10.3	4.8	11	1.4	12.3
88	38	12	5.7	12.3	1.7	13.3
132	66	8.3	6.0	11	1.4	11
176	97	7.7	6.6	12.7	1.9	13

Table 3 - Random Resultants

n	S	B	I	B	D	B
8	5.4	8	2.7	8	0.86	11
18	179	10	9.6	10	4.1	18

Table 4 - Random Products, b = 44

n	S	B	I	B	D	B
5	4.7	13	3.3	16	0.75	13
10	26	12	6.3	15	1.48	13
15	85	13	20.1	31	3.1	18

Table 5 - Chebyshev Polynomials

n	S	B	I	B	D	B
5	2.9	12	2.2	14	0.68	14
10	10.9	11	3.6	11	0.83	11
15	25	10	7.1	14	1.73	13

6. Conclusion

We have shown that two new algorithms are theoretically and practically faster than previous known algorithms, based on Sturm sequences. We expect this also being true compared to Zassenhaus' method of indexing because Kempfert's algorithm relies also on Sturm sequences. The main attraction to study the sign determination for real algebraic numbers is its application for real root isolation of real algebraic polynomials [8]. In particular the real root isolation algorithms based on derivative sequences or Descarte's sign rule make havily use of sign calculations and can easily be extended from $Z[x]$ to $Q(\alpha)[x]$. In contrast, the calculation of Sturm sequences over $Q(\alpha)$ is even worse than over Z.

References

[1] G. E. Collins, Computer Algebra of Polynomials and Rational Functions, Amer. Math. Monthly 80, (Aug.-Sept. 1973), 725-754

[2] C. R. Rubald, Algorithms for Polynomials over a Real Algebraic Number Field, Computer Sciences Dep., University of Wisconsin, Madison Techn. Report No. 206, Jan. 1974

[3] H. Zassenhaus, A Real Root Calculus, Proceedings of a Conference held at Oxford, (Aug.-Sept. 1967), 383-393

[4] L. E. Heindel, Integer Arithmetic Algorithms for Polynomial Real Zero Determination, J. ACM, 18 (Oct.1971), 533-548

[5] G. E. Collins, R. Loos, Polynomial Real Root Isolation by Differentiation, Proceedings of 1976 ACM Symposium on Symbolic and Algebraic Computation

[6] G. E. Collins, A. Akritas, Polynomial Real Root Isolation Using Descarte's Rule of Signs, Proceedings of 1976 ACM Symposium on Symbolic and Algebraic Computation

[7] R. Loos, private communication

[8] S. Rump, Diplomarbeit, Kaiserslautern 1976

[9] G. E. Collins, a list of SAC-1 reports is contained in the KWIC-Index, SIGSAM Bulletin of the ACM, 8 (1974), 17-44

[10] G. E. Collins, Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition, Lecture Notes in Computer Science, Vol. 33, pp. 134-183, Springer Verlag, Berlin, 1975

[11] H. Kempfert, On Sign Determinations in Real Algebraic Numbers Fields, Num. Math. 11, (1968) 170-174

[12] M. Mignotte, An Inequality About Factors of Polynomials, Mathematics of Computation, Vol. 28, No. 128 (October 1974), pp. 1153-1157

[13] M. Mignotte, Sur la complexité certains algorithmes ou intervient la séparation des racines d'un polynôme

[14] G. E. Collins and E. Horowitz, The Minimum Root Seperation of a Polynomial, Math. of Comp., Vol. 28, No. 126(1974) 589-597

Algorithms for Symmetrical Polynomials

Emi Lauer
University of Kaiserslautern

March 1976

Abstract

A special representation for symmetrical polynomials is introduced. Algorithms for the ring operations and for several versions of Gauss' method to express an arbitrary symmetrical polynomial by the elementary symmetrical functions are given and analyzed. Empirical observations show that the representation which is the most economical in terms of space is not always the most economical in terms of time.

1. Introduction

Symmetrical functions play an important rôle in algebra and its applications. In particular, many proofs in the theory of algebraic numbers refer to them explicitly. Resultants [1] and Sturm sequences [2] can be generated as symmetrical functions. In practice, problems often are symmetrical in some if not all variables and one may ask how the general algorithms of a Computer Algebra system can take advantage of the additional structure. A special set for symmetrical functions could even be helpful if only s out of r variables are symmetrical. In the case of polynomials one can separate out the symmetrical variables, say $x_1,\ldots,x_s$, from the rest $x_{s+1},\ldots,x_r$, $2 \leq s < r$, and perform polynomial operations in $D_s[x_{s+1},\ldots,x_r]$ with $D_s = D[x_1,\ldots,x_s]$ using symmetrical arithmetic in the coefficient domain D_s.

As a first step in this direction we have implemented an algorithm due to Gauss (section 2), an efficient algorithm for the generation of the elementary symmetrical polynomials (section 3), and algorithms for arithmetical operations (section 4) where we also study the impact of a space saving representation on the computing times. Finally (section 6) we give some empirical results for different realizations of Gauss' algorithm and the arithmetic.

2. The Fundamental Theorem and the Gauss-Algorithm

The fundamental theorem of symmetrical functions is of interest because of a constructive proof due to Gauss [1]. His proof is easily understood in terms of a distributive canonical polynomial representation which are actually used in algebra systems like ALTRAN [3]. A non-zero polynomial in $r > 0$ variables is an ordered sequence of terms, each term consisting of a non-zero numerical coefficient and a monomial $x_1^{n1}\ldots x_r^{nr}$. The ordering is lexicographically decreasing on the exponent vectors $(n_1,\ldots,n_r)$.

Fundamental Theorem

Let $P \in Z[x_1,\ldots,x_r]$ be a symmetrical polynomial. Then there exists a polynomial $Q \in Z[y_1,\ldots,y_r]$ such that $P(x_1,\ldots,x_r) = Q(\sigma_1,\ldots,\sigma_r)$ where σ_i is the i-th elementary function in the r variables $x_1,\ldots,x_r$.

Gauss proved that the following algorithm produces the polynomial Q in finitely many steps.

Q ← GAUSS(P)

[The notations are those of the Fundamental Theorem.]

(1) Construct $\sigma_1,\ldots,\sigma_r$ and set Q = 0.

(2) while $P \neq 0$ do the following

{let $a\,x_1^{n_1}\ldots x_r^{n_r}$ be the lexicographically first term of P;

Then set $P_1 = a\,\sigma_1^{n_1-n_2}\ldots\sigma_{r-1}^{n_{r-1}-n_r}\sigma_r^{n_r}$,

$Q_1 = a\,y_1^{n_1-n_2}\ldots y_{r-1}^{n_{r-1}-n_r}y_r^{n_r}$,

$P = P - P_1$

$Q = Q + Q_1$}

and stop. ∎

First, one notes that the exponents $n_1-n_2,\ldots,n_{r-1}-n_r, n_r$ of P_1 are all non-negative because the symmetrical polynomial P contains with the first term

$a \cdot x_1^{n_1} \cdots x_r^{n_r}$ also the sum $a \cdot \Sigma\, x_1^{n_{i_1}} \cdots x_r^{n_{i_r}}$ where $(n_{i_1},\ldots,n_{i_r})$ runs over all different permutations of $(n_1,\ldots,n_r)$. We call this sum a symmetrical component of P and denote it by $S(a;n_1,\ldots,n_r)$. For the lexicographically first term one has therefore $n_1 \geq n_2 \geq \ldots \geq n_r$.

Secondly, the polynomial $P_1 = a\,\sigma_1^{n_1-n_2} \cdots \sigma_r^{n_r}$ has the same leading term (and hence component) as P, since $x_1^e \ldots x_i^e\, x_{i+1}^o \ldots x_r^o$ is the leading term of σ_i^e. Therefore, the subtraction $P-P_1$ removes the leading component from P (but may introduce new components, which are lexicographically later, from P_1). Finally, the algorithm terminates since there are only a finite number of components whose lexicographically first term is later than the lexicographically first term of P. ∎

In fact, a symmetrical polynomial with leading component $S(a;n,\ldots,n)$ has at most $A(n,r) = \binom{n+r}{r}$ symmetrical components. Using classical polynomial arithmetic in step 2 one has:

Theorem 1 The computing time of the Gauss-algorithm is dominated by

$$2^r \cdot r^2 \cdot (n+1)^{r+3} \cdot \binom{n+r}{r}^2 \cdot L(d_p)\,,$$

where $L(d_p)$ is the integer length of the maximum norm d_p of P.

Empirical computing times of the Gauss-algorithm using classical arithmetic are given in Table 3, column G_1, in section 6.

3. Generation of the Elementary Symmetrical Functions

Let $\sigma_i^{(s)}$ be the i-th elementary symmetrical function in s variables. Then the following recurrence relation holds

$$\sigma_i^{(s)} = x_s\, \sigma_{i-1}^{(s-1)} + \sigma_i^{(s-1)}, \text{ for } i = 1,\ldots,s$$

with

$$\sigma_o^{(o)} = \sigma_o^{(s)} = 1 \text{ and } \sigma_s^{(s-1)} = 0, \text{ for all } s \geq 1.$$

$\sigma_i^{(s)}$ is a linear polynomial in x_s having coefficients $\sigma_{i-1}^{(s-1)}$, $\sigma_i^{(s-1)}$ in s-1 variables. Using a recursive polynomial representation an algorithm based on the recurrence relation generates inductively the list $(\sigma_o^{(s)},\ldots,\sigma_s^{(s)})$ by taking adjacent pairs from a list $(\sigma_o^{(s-1)},\ldots\sigma_{s-1}^{(s-1)})$. If the polynomials are represented by lists allowing overlapping $\sigma_i^{(s)}$ is constructed from $\sigma_{i-1}^{(s-1)}$ and $\sigma_i^{(s-1)}$ in constant time and space. The construction of the list takes $c \cdot s$ operations and $d \cdot s$ list cells for some constants c and d. Since $0 \leq s \leq r$ we have the

Theorem 2. The algorithm to generate $\sigma_o^{(r)},\ldots,\sigma_r^{(r)}$ (which altogether contain 2^r terms) requires $\sim r^2$ time and space.

$\sigma_j^{(r)}$ is a polynomial whose exponent vectors contain j ones and (r-j) zeros. So, constructing all bit strings of length r with j ones, j running from 0 to r, the total number of terms is 2^r. In this case a distributive representation would have been inferior. The algorithm is simple to implement and the observed times are very small (see table 1).

4. Arithmetic for a Special Representation of Symmetrical Polynomials

By a consideration of Gauss' proof one sees that a symmetrical polynomial is a sum of symmetrical components. Recalling that $S(a;n_1,\ldots,n_r)$ denotes the sum of all different permutations of monomials with coefficient a and exponent vectors $(n_{i_1},\ldots,n_{i_r})$ this sum consists of maximally r! terms for the case that all n_i, n_j $(i \neq j)$ are different. Obviously, a representation more economical in space would be obtained by representing each symmetrical component by its leading term and suppressing all other permuted terms.

Hence, if $P = \sum_{i=1}^{s} S(a_i;n_1^{(i)},\ldots,n_r^{(i)})$, $n_1^{(i)} \geq \ldots \geq n_r^{(i)}$, is a symmetrical polynomial, we define the component representation of P as the list $(a_1,(n_1^{(1)},\ldots,n_r^{(1)}),\ldots,a_s,(n_1^{(s)},\ldots,n_r^{(s)}))$, where the exponent vectors are lexicographically decreasing. In the worst case a symmetrical polynomial of degree n in r variables has in the general representation $(n+1)^r$ terms but only $\binom{n+r}{r}$ symmetrical components. Asymptotically the saving ratio goes for fixed r as 1/r! and for fixed n to 0 in favor of the representation by components.

For compatibility reasons we need conversion algorithms for the two representations. If a symmetrical polynomial P is given in recursive representation, one obtains the component representation by selecting from all exponent vectors of P those which are in decreasing order. This can be done recursively:

let $P_i\ (x_1, \ldots, x_{r-1})$ be the coefficient of $x_r^{e_i}$ and $L_i = (a_1^{(i)}v_1^{(i)}, \ldots a_k^{(i)}v_k^{(i)})$ the list of decreasing exponent vectors of P_i together with the corresponding integer coefficients. Then if $v_j^{(i)} = (n_1^{ij},\ldots, n_{r-1}^{ij})$ and $n_1^{ij} \leq e_i$, the augmented exponent vector $(e_i, n_1^{ij},\ldots, n_{r-1}^{ij})$ belongs to the component representation of P. The recursion base is r=1 in which case both representations coincide.

The conversion from component to recursive representation is even easier, since the recursive representation of the terms $a\ x_1^{n_1} \ldots x_r^{n_r}$ can be constructed by simple list - processing. In order to obtain $S(a, n_1,\ldots, n_r)$ in recursive representation one has to construct the monomials $a\ x_1^{n_{i_1}} \ldots x_r^{n_{i_r}}$ for all different permutations of the exponent vector and add them with usual arithmetic.

The important question is how the polynomial algorithms react to the drastic drop in redundant information in the data structure of symmetrical polynomials. Obviously things are not as simple as the (computer folklore) statement suggests that the gain in space has to be paid by a loss in time and vice versa.

The first algorithms we consider are polynomial addition and equivalently negation and subtraction. Polynomial addition is essentially a merging of terms, the cost being directly proportional to the number of terms of the inputs. In the component representation addition is a merging of components represented by leading terms, the cost being reduced by the number of dropped terms. Table 2 (a,b) in section 6 compares the addition of symmetrical polynomials in the component representation and in the general representation. The asymptotic maximum computing times are $\binom{n+r}{r}(r+L(d))$ for the component representation as compared with $(n+1)^r L(d)$ for the recursive representation where n is the maximum of the degrees and d the maximum norm of the inputs.

Next we consider the multiplication of symmetrical polynomials in component representation. In the example

$$A = S(3;2,1,0) = 3(x^2y+x^2z+xy^2+xz^2+yz^2+y^2z)$$

$$B = S(-2;2,0,0) = -2(x^2+y^2+z^2)$$

we have

$$C = A*B = -6(x^4y+x^4z+x^3y^2+x^3z^2+x^2yz^2+x^2y^2z+$$
$$+x^2y^3+x^2y^2z+xy^4+xy^2z^2+y^3z^2+y^4z+$$
$$+x^2yz^2+x^2z^3+xy^2z^2+xz^4+yz^4+y^2z^3) =$$
$$= S(-6;4,1,0)+S(-6;3,2,0)+S(-12;2,1,1)$$

and the question arises whether there is a multiplication algorithm for symmetrical components in order to get the right hand side from the left hand side directly.

We define the notion of the star-component by

$$S^*(n_1,\ldots,n_r) = \sum_{\pi\in S_r} x_1^{n_{\pi(1)}} \ldots x_r^{n_{\pi(r)}},$$ where S_r denotes the set of all permutations of $\{1,\ldots,r\}$, the symmetric group. Recall that for $S(a;n_1,\ldots,n_r)$ the summation runs over all different permutations of $(n_1,\ldots,n_r)$ resulting for the case of equal n_i, n_j's $(i \neq j)$ in fewer terms than S^*. However there is a simple relation between S and S^*. We define the density $\rho(n_i)$ of n_i as the frequency of the occurrence of n_i in the final segment $(n_i, n_{i+1},\ldots,n_r)$ of the exponent vector. The repetition factor of an exponent vector is defined by

$$rep(n_1,\ldots,n_r) = \rho(n_1)\ldots\rho(n_r).$$

The relation between S and S^* is given by the

Lemma a $S^*(n_1,\ldots,n_r) = rep(n_1,\ldots,n_r)\cdot S(a; n_1,\ldots,n_r)$.

Finally, the modified permanent of a r x r matrix (a_{ij}) is defined by $per(a_{ij}) = \sum_{\pi\in S_r} S^*(a_{1,\pi(1)},\ldots,a_{r,\pi(r)})$.

(In a "true" permanent $S^*(a_{1,\pi(1)},\ldots,a_{r,\pi(r)})$ is replaced by the product $\prod_{i=1}^{r} a_{i,\pi(i)}$).

Theorem 3 (Loos)

The product of two symmetrical components is given by

$$S(a;m_1,\ldots,m_r) * S(b;n_1,\ldots,n_r) = \frac{a\cdot b}{rep(m_1,\ldots,m_r)\ rep(n_1,\ldots,n_r)} per(m_i+n_j).$$

For brevity we omit the simple proof.

In the example above we had

$$S(3;2,1,0)*S(-2;2,0,0) =$$

$$= \frac{3 \cdot (-2)}{\text{rep}(2,1,0)\ \text{rep}(2,0,0)} \text{ per} \begin{bmatrix} 4 & 2 & 2 \\ 3 & 1 & 1 \\ 2 & 0 & 0 \end{bmatrix} =$$

$$= -\frac{6}{2}[2S*(4,1,0)+2S*(3,2,0)+2S*(2,1,1)] =$$

$$= S(-6;4,1,0)+S(-6;3,2,0)+S(-12;2,1,1)$$

The product algorithm for two symmetrical polynomials A and B represented by sequences of symmetric components processes in the outer loop each component of A and multiplies it in the inner loop with each component of B and the results are added.

If the modified permanent is expanded recursively with respect to the first column it will usually contain exponent vectors which are not in decreasing order. These have to be sorted since the component representation requires decreasing exponent vectors. Likewise the list of exponent vectors which is obtained after this sorting is generally not in lexicographically decreasing order and must be sorted, too. Finally the coefficents of equal exponent vectors must be added.

This twofold sorting process and the summing up of equal vectors slows down the product algorithm considerably such that the gain in terms is overcompensated. Although we could multiply symmetric polynomials which could not be held in memory in the general representation the time comparisons were in disfavor for the component representation in all cases. Table 2 (a,b) of section 6 gives the times for component arithmetic. Advantages of the general algorithm are that in multiplying terms only single terms are produced but not polynomials and therefore polynomial addition has not to be done in the inner loop. We expect that in our algorithm the inner loop with the component product can be improved.

Theorem 4

Let t_A, t_B denote the number of components of A and B respectively. Then the time to multiply A and B is dominated by

$$t_A t_B \binom{m+n+r}{r} r! r^2 \log(r) L(|A|_\infty) \cdot L(|B|_1)$$

where r is the number of variables, $m = \deg_r(A)$, $n = \deg_r(B)$.

For the important case of the Gauss-algorithm the component product algorithm can be designed more satisfactory. The only products of the Gauss-algorithm are products of elementary symmetrical functions, and for the case of one factor being of this special kind the modified permanent according to Theorem 3 can be produced in sorted form, both the exponents in the vectors and the vectors in the permanent.

We recall that $\sigma_j^{(r)}$ consists of the single component $S(1;1,\ldots,1,0,\ldots,0)$ with j 1's and r-j 0's after the coefficient. Then the matrix of $S(a;n_1,\ldots,n_r) * \sigma_j$ has the special form: the first j rows are $(n_1+1, \ldots, n_r+1) = R_1$, the last (r-j) rows are $(n_1,\ldots,n_r) = R_2$. An element of the modified permanent is a vector consisting of j elements of R_1 and the r-j complementary elements of R_2; this means that it is just the componentwise sum of $(n_1,\ldots,n_r)$ and a permutation of $S_j = (1,\ldots,1,0,\ldots,0)$. The complete modified permanent is obtained by doing this for all $\binom{r}{j}$ permutations of S_j. If one takes these permutations in lexicographically decreasing order one can prove that:

(1) all resulting vectors which do not have decreasing exponents appear already as ordered sum vectors of another earlier permutation of S_j and therefore can be left out if one takes into account the multiplicity of the dropped vectors.

(2) the list of remaining ordered vectors is lexicographically ordered .

The multiplicity can be computed in the following way: let $(n_1,\ldots,n_r)$ contain k adjacent equal elements; let some permutation of S_j contain e ones in this segment. Then there exist exactly $\binom{k}{e}$ permutations of S_j which permute only this segment and let all other elements fixed. Since $n_{i+1} = \ldots = n_{i+k}$, all these permutations give permutations of the same sum vector and hence only one of those (the sorted one) is taken; its multiplicity is therefore $\binom{k}{e}$. If $(n_1,\ldots,n_r)$ contains several such blocks of equal elements, the total multiplicity is the product of the multiplicities of each block.

Since only the different permutations are considered, the numerical coefficients must be multiplied with a correction factor $r!/\binom{r}{j} = j! \cdot (r-j)!$, but this is exactly the repetition factor of S_j and hence cancels.

Example:

Compute $S(-7; 5,5,5,4,3,3) * \sigma_2$

The $\binom{6}{2}$ different permutations of S_2 are:

1.)	(1,1,0,0,0,0)	9.)	(0,1,0,0,0,1)
2.)	(1,0,1,0,0,0)	10.)	(0,0,1,1,0,0)
3.)	(1,0,0,1,0,0)	11.)	(0,0,1,0,1,0)
4.)	(1,0,0,0,1,0)	12.)	(0,0,1,0,0,1)
5.)	(1,0,0,0,0,1)	13.)	(0,0,0,1,1,0)
6.)	(0,1,1,0,0,0)	14.)	(0,0,0,1,0,1)
7.)	(0,1,0,1,0,0)	15.)	(0,0,0,0,1,1)
8.)	(0,1,0,0,1,0)		

The permutations 2,5,6,7,8,9,10,11,12,14 produce sum vectors which are not in decreasing order .

The others give :

1.) : (6,6,5,4,3,3) with multiplicity 3
3.) : (6,5,5,5,3,3) " 3
4.) : (6,5,5,4,4,3) " 6
13.) : (5,5,5,5,4,3) " 2
15.) : (5,5,5,4,4,4) " 1

The numerical coefficient of (6,5,5,5,3,3) for example is

$$\frac{(-7)\cdot \mathrm{rep}(6,5,5,5,3,3)\cdot 3}{\mathrm{rep}(5,5,5,4,3,3)} = -21 .$$

So we finally get:

$$S(-7;\ 5,5,5,4,3,3) \times \sigma_2 =$$
$$S(-7;\ 6,6,5,4,3,3) + S(-21;\ 6,5,5,5,3,3) +$$
$$+S(-14;6,5,5,4,4,3) + S(-28;\ 5,5,5,5,4,3) +$$
$$+S(-21;5,5,5,4,4,4).$$

These observations are the base of the following algorithm:

L ← SSCPR(C,j)

[Special symmetrical component product. $C = (c;\ n_1,\ldots,n_r)$ a symmetrical component, $L = C \times \sigma_j^{(r)}$.]

(1) [Initialize]

Set $N = (n_1,\ldots,n_r)$, $S = (1,\ldots,1,0,\ldots,0)$ with j 1's, and L = ( ).

(2) [Permutation loop]

For all different permutations of S perform step 3, 4 and 5 and stop.

(3) [Process single permutation]

i ← 1; m ← 1; [multiplicity of retained component] while i < r do
{k ← 1; [number of adjacent equal elements in N]
e ← s_i; [number of 1's in this segment of $S=(s_1,\ldots,s_r)$]
while i < r & $n_i = n_{i+1}$ do
[process block]
{if $s_i < s_{i+1}$ then process next permutation, if any;
k ← k + 1;
e ← e + s_{i+1};
i ← i + 1};
m ← m × $\binom{k}{e}$;
i ← i + 1}.

(4) [Numerical coefficient]

d ← c × m × rep(N+s)/rep(N).

(5) [Join to output] Suffix (d,N+S) to L ∎

Computing time $\prec \binom{r}{j}(r+L(c))$, if r is a small single precision number.

Theorem 5:

The computing time of the Gauss-algorithm using SSCPR is dominated by
$2^r \cdot r \cdot n^2 \cdot (n+1)^r \cdot \binom{n+r}{r}^3 \cdot L(|P|_\infty)$.

5. The Bratley-McKay Algorithm

In order to obtain empirical comparisons with existing algorithms we implemented an algorithm of Bratley and McKay which is published as Algorithm 305 in the Collected Algorithms from CACM [5].

This algorithm expresses the symmetrical component $S(1;\ n_1,\ldots,\ n_r)$ as a sum of determinants in the elementary symmetrical functions. The symmetrical component is first expressed in terms of Schur functions which are then evaluated as determinants in the elementary symmetrical functions.
Our implementation is just a transliteration of the ALGOL program to ANSI-FORTRAN with the modifications that lists are used instead of arrays, that the elementary symmetrical functions are given as symbolic polynomials (not as values of these polynomials), and the determinants (which have polynomial entries now) are computed by use of the SAC-1 algorithm PDET.
Unfortunately there exists no theoretical computing time analysis of this algorithm. The empirical times are given in table 3, column BM.

6. Empirical results

In the following some observed computing times are presented for the generation of the elementary functions, the arithmetic of symmetric polynomials and the Gauss algorithm with different arithmetic. All times are given in seconds and were obtained by using SAC-1 [4] on the University of Kaiserslautern Telefunken TR 440 computer.

In a forthcoming Diplomarbeit by the author, supervised by Professor R. Loos, the omitted proofs, algorithm descriptions and programs will be given.

TABLE 1: Generation of the elementary symmetrical functions in r variables.

r	EGEN
5	0.0051
10	0.0169
15	0.0354
20	0.0595

TABLE 2: Arithmetic with symmetrical polynomials

Random symmetrical polynomials in r variables and of degree n in each variable; maximum norm bounded by 2^8; half dense (50 % of all possible symmetrical components) PSUM, PPROD: usual SAC-1 algorithm SPSUM, SPPROD: special algorithms for symmetrical polynomials.

(a)

r	n	PSUM	SPSUM	PPROD	SPPROD
2	2	0.006	0.004	0.06	0.10
2	4	0.009	0.005	0.36	0.85
2	6	0.019	0.012	1.67	6.05
2	8	0.030	0.019	4.55	23.0
4	2	0.054	0.010	6.28	5.95
4	4	0.295	0.040	+)	336.1
4	6	0.905	0.113		
4	8	+)	0.264		

+) memory overflow

(b)

r	n	PSUM	SPSUM	PPROD	SPPROD
2	2	0.006	0.005	0.06	0.10
4	2	0.054	0.008	7.18	6.20
6	2	0.458	0.015	+)	1075.0
3	4	0.052	0.014		
4	4	0.282	0.037		
5	4	+)	0.074		

+) memory overflow

TABLE 3: Gauss´ algorithm with different arithmetic

G_1: with usual arithmetic

G_2: with SPSUM and SPPROD

G_3: with SSCPR

BM: with Bratley McKay algorithm; Random polynomials like in TABLE 2, maximum norm < 2^4

r	n	G_1	G_2	G_3	BM
3	2	0.46	0.57	0.12	0.54
3	4	5.91	4.94	1.63	18.00
4	2	3.18	4.66	0.39	1.93
4	4	55.30	53.56	5.13	112.05
5	2	9.94	42.92	0.71	4.73
5	4	437.9	707.6	17.0	728.12

References

[1] B. van der Waerden, Algebra I, Heidelberg

[2] E. Netto, Vorlesungen über Algebra, Leipzig, 1896, Bd.I

[3] W.S. Brown, ALTRAN User´s Manual, Bell Laboratories, Murray Hill, 1973

[4] G.E. Collins, a list of SAC-1 reports is given in the KWIC-Index in SIGSAM Bulletin of the ACM, New York, 8, (1974), 17-44

[5] P. Bratley and J.K.S.McKay,Algorithm 305: Symmetric polynomials,Comm.ACM, VOL.10,(July 1967),450; 11,(April 1968) 272.

Algebraic Algorithms using p-adic Constructions

David Y. Y. Yun
IBM T. J. Watson Research Center
Yorktown Heights, New York

1. Introduction

Machine computation of many algebraic operations of interest and utility to mathematicians and scientists has received increasing attention in the recent years. A variety of terminology has been used to describe this area of mathematical computation. Among these are symbolic manipulation, computer algebra, algebraic methods, constructive algebra, and seminumerical algorithms. Whatever name is chosen for the field currently known as symbolic and algebraic manipulation, the most fundamental part of any symbolic computer software system for the actual computing is always the workhorse subpackage that deals with polynomial and rational functions.

Polynomials, quotients of which are defined to be rational functions, are basic algebraic quantities with numerous interesting and useful properties. Their degree, the finite number of non-zero terms, their divisibility and division procedure, their primeness or irreducibility, and their unique factorizations are all essential properties in their representation and computation. Many of these are contained in the definitions for spaces of polynomials, such as polynomial rings, commutative rings, integral domains, unique factorization domains (u.f.d.), and Euclidean domains. The domain of coefficients determines what kind of space the polynomials will form which in turn affects the kind of computational methods applicable for the particular application. Historically, constructive procedures for polynomials have received considerable attention due to the inherent importance, varying properties, and intriguing complexity of polynomials. The generalization of classical methods, such as the Euclidean algorithm or the Chinese remainder theorem, from the integers, Z, to polynomials, e.g., Z[x], has introduced numerous computation (and theoretical) problems. Researchers in algebraic algorithms have responded with a variety of answers or solutions to these problems so that the advancement of algebraic computation is in continuous progress.

Recently, several advances in algebraic computations have been made. These include polynomial and power series division; power series inversion and reversion polynomial factorization and greatest common divisor (GCD) calculation; and partial fraction decomposition of rational functions. The constructive methods for these algebraic operations appear to have little relationship among themselves. However, as we will show in this paper, the classical Newton's method plays an overwhelming role in unifying the derivation of these algebraic constructions. In addition, the appropriate generalization of the p-adic representation of integers to polynomials plays a fundamental role in providing the basic canonical form, an approximation space over which the algebraic Newtonian approach iterates. Through this unifying representation and derivation, we also demonstrate the potential of discovering, designing, and synthesizing new algebraic algorithms.

2. P-adic Representation and Approximation

We first introduce the notion of a "p-adic" representation of integers and of polynomials. Given any prime p in Z, each integer a in Z can be considered to have a unique p-adic representation

$$a = \sum_{i=o}^{k} a_i p^i, \quad |a_i| \le p/2, \ k \text{ is finite.}$$

Correspondingly, any polynomial F(x) in Z[x] has a unique p-adic representation

$$F(x) = \sum_{j=o}^{d} F_j x^j \qquad (F_j \text{ in } Z,\ F_d \neq 0,\ d \text{ is finite})$$

$$= \sum_{j=o}^{d} \Big(\sum_{i=o}^{k_i} F_{ij} p^i \Big) x^j \qquad \text{(by putting each } F_j \text{ in p-adic form)}$$

$$= \sum_{i=o}^{k} \Big(\sum_{j=o}^{d} F_{ij} x^j \Big) p^i \qquad (k = \max_i k_i,\ F_{ij} = 0 \text{ for } k_i < i \le k)$$

where it is a series in p^i with coefficients which are polynomials in $Z_p[x]$.

Let Z[[x]] be the space of finite p-adic series in the indeterminate x over the integers. It is easy to show that the domain of polynomials in x over the integers, Z[x], is isomorphic to Z[[x]]. Thus, each finite p-adic series is in one-to-one correspondence to a polynomial in Z[x] so that it can be used as a canonical representation of that polynomial. This view point leads naturally to the notion of "p-adic approximation" of polynomials in Z[x]. That is, any polynomial in Z[x] can be approximated by a sequence of elements from Z[[x]]. The sequence is finite, i.e., the approximation will eventually be exact, since polynomials have bounded coefficients (k is finite).

As an example, let p = 5. Then for

$$F(x) = 19x^5+5x^4-48x^2+31x-1$$
$$= (-1+(-1)\cdot 5+1\cdot 5^2)x^5+(0+1\cdot 5)x^4+0\cdot x^3$$
$$+ (2+0\cdot 5+(-2)\cdot 5^2)x^2+(1+1\cdot 5+1\cdot 5^2)x^1+(-1)x^0$$
$$= (-1x^5+2x^2+1x-1)5^0+(-1x^5+1x^4+1x)5^1$$
$$+ (1x^5-2x^2-1x)5^2,$$

the first term, $(-x^5+2x+x-1)$, is in $Z_5[x]$ which is contained in Z[[x]] and is considered to be the "first order p-adic approximation" to F(x); the first two terms, $(-x^5+2x^2+x-1)+(-x^5+x^4+x)\cdot 5$ which is in $Z_{25}[x]$ also contained in Z[[x]], is considered the "second order p-adic approximation" to F(x); all three terms which is in $Z_{125}[x]$ and, in this case, equal to the polynomial F(x) is the "third order p-adic approximation" to F(x).

To formalize this notion of approximation, we make the following definitions:

Definition 2.1: F(x) and G(x) in Z[[x]] are congruent modulo p^m, written as $F(x) \equiv G(x) \pmod{p^m}$, if $F_i(x) = G_i(x)$ in $Z_p[x]$ for all i=0,1,...,m-1 (i.e., $F_i(x) \equiv G_i(x) \pmod p$ for all i).

Definition 2.2: The order of F(x) in Z[[x]] is m if $F_m(x) \neq 0$ and $F_i(x) = 0$ for all $i < m$. (i.e., $F_m(x)p^m$ is the first non-zero term in the p-adic series).

Definition 2.3: G(x) is an order m approximation to F(x) in Z[[x]] if $F(x) \equiv G(x) \pmod{p^m}$, (i.e., order of F(x) - G(x) $\geq$ m.)

Definition 2.4: The error in approximating F(x) by G(x) in Z[[x]] is F(x) - G(x).

As an example of this p-adic approximation, we refer back to the Hensel Lemma of 1913 [HEN13], which is an algebraic computational technique usually associated with number theory and p-adic analysis.

Hensel Lemma: Let p be a prime in Z and F(x) be a given polynomial in Z[x]. Let $G_1(x)$ and $H_1(x)$ be two relatively prime polynomials in $Z_p[x]$ such that $F(x) \equiv G_1(x)\, H_1(x) \pmod p$. Then for any integer k > 1, there exist polynomials $G_k(x)$ and $H_k(x)$ in $Z_q(x)$, where $q = p^k$, such that $F(x) \equiv G_k(x)\, H_k(x) \pmod q$ and $G_k \equiv G_1 \pmod p$, $H_k \equiv H_1 \pmod p$.

The constructive proof of this lemma provides a method for lifting factors of a polynomial over a finite field (Z_p [BER67]) step-by-step toward a larger subdomain of Z[x], while preserving the relevant properties, and eventually toward the desired result in Z[x]. The correspondence to p-adic approximation of the factors can readily be seen. A variation of the Hensel Lemma was proposed by Zassenhaus [ZAS69].

Zassenhaus Construction: Let p, F(x), $G_1(x)$, and $H_1(x)$ be given as before in the Hensel Lemma. For any integer k>1, there exist polynomials $G_k(x)$ and $H_k(x)$ in $Z_q[x]$, where $q = p^{2^{k-1}}$, such that $F(x) \equiv G_k(x)\, H_k(x) \pmod q$ and $G_k \equiv G_1 \pmod p$, $H_k \equiv H_1 \pmod p$. Since the modulus increases quadratically as k increases, the order of approximation doubles each time, hence it is a quadratically convergent method.

Both Hensel's and Zassenhaus' constructions are approximation methods in Z[[x]] in essence where the desired results of the computation are given to more and more correct terms of their p-adic representations at each step of the constructions. We will derive these algebraic methods as well as other algorithms using the familiar analytical technique of Newton's iteration formula.

3.0 Newton's Iteration

In its most readily recognizable form, Newton's method provides an iteration formula

$$x_{k+1} = x_k - f(x_k)/f'(x_k)$$

for finding a root of some given differentiable function f(x). One familiar derivation of this formula is via the first order Taylor series expansion of f about some point x_k:

$$f(x) = f(x_k) + f'(x_k)\,(x-x_k).$$

Assuming x is, or is sufficiently near, a root of f so that f(x) = 0, and solving for x, we have

$$x = x_k - f(x_k)/f'(x_k),$$

hence the Newton's iterative formula.

3.1 A Sample Usage in Algebraic Computation

Now we apply the derivation techniques for the classical Newton's iteration formula on the space of p-adic series Z[[x]] to get a Newtonian algebraic construction for polynomial root-finding. Let Φ be a mapping from Z[[x]] into Z[[x]]. For a root, G(x) in Z[[x]], of Φ, the first order Taylor series expansion of Φ at G is

$$0 = \Phi(G) = \Phi(G_k) + (G-G_k)\,\frac{d\Phi(G_k)}{dG}.$$

Now we specify $\Phi(G(x)) = F-G^n$ for a given F(x) in Z[[x]], so that the zero of Φ is the nth root (if it exists) of the polynomial F(x), i.e., $F = G^n$. Define $\Delta G_k = G-G_k$, then ΔG_k is the kth update necessary to compute the next approximation by the formula of

$$G_{k+1} = G_k + \Delta G_k.$$

The p-adic approximation method for finding the nth root of the polynomial F(x) is, then, solve $F-G_k^n = (\Delta G_k)\cdot n\cdot G_k^{n-1}$ at each step of the Newtonian iteration.

Let G_1 be a given "initial guess" which approximates $\Phi(G) = 0$ in the sense that $\Phi(G_1) \equiv 0 \pmod p$, i.e., $F(x) \equiv G_1^n(x) \pmod p$, G_1 in $Z_p[x]$. Inductively, on the kth step and for some power of p, q_k, assume $G_k(x)$ has been found such that

$$\Phi(G_k) = F(x) - G_k(x)^n \equiv 0 \pmod{q_k} .$$

Then the equation to be solved is actually

$$F - G_k^n \equiv (\Delta G_k)\cdot n\cdot G_k^{n-1} \pmod{q_{k+1}}.$$

The introduction of the modulus q_{k+1} is justified because the (inductive) goal is to get $G_{k+1}(x)$ in Z[[x]] such that

$$\Phi(G_{k+1}) = F - G_{k+1}^n \equiv 0 \pmod{q_{k+1}}.$$

But now the Newtonian iteration can be altered to that of solving for $C_k(x)$ $(=\Delta G_k/q_k)$ in the equation

$$\frac{F-G_k^n}{q_k} = C_k\cdot n\cdot G_k^{n-1} \pmod{q_{k+1}/q_k}.$$

Then use the update formula of

$$G_{k+1} = G_k + q_k\cdot C_k.$$

We point out here that each unknown $C_k(x)$ is in the domain $Zq_{k+1}/q_k[x]$ which is a more structured subdomain of Z[x] or Z[[x]]. In fact, if we now specify q_k to be p^k, then a "linearly convergent" version of the p-adic approximation method results in the following form:

Given the initial p-adic approximation $G_1(x)$ in $Z_p[x]$.

For all $k \geq 1$, solve the equation

$$\frac{F-G_k^n}{p^k} = C_k\cdot n\cdot G_1^{n-1} \pmod p \text{ for } C_k(x) \text{ in } Z_p[x]$$

and get the next approximation by

$$G_{k+1} = G_k + p^k C_k .$$

Note the simplicity of the equation to be solved, i.e., it involves only $G_1(x)$ for all k, while the computation of $C_k(x)$ is simply a division operation of the known left-hand side by the fixed element $n\cdot G_1(x)$ in the Euclidean domain $Z_p[x]$. Thus, as long as $G_1(x)$ in $Z_p[x]$ can be found such that $F \equiv G_1^n \pmod p$ and G_1 is an invertible element of $Z_p[x]$, the above scheme can be used to compute the nth root of F(x). Whenever any C_k can not be computed by dividing nG_1 exactly into the left-hand side, the algorithm terminates with the indication that n^{th} root of F does not exist. But if the unique n^{th} root of F does exist (say G) then G_1 exists in $Z_p[x]$, each C_k can be found by division, and the p-adic approximation algorithm must terminate due to the finiteness of the solution G(x).

We may also specify q_k to be $p^{2^{k-1}}$ which will lead to a "quadratically convergent" method as follows:

Given the initial p-adic approximation $G_1(x)$ in $Z_p[x]$.

For all $k \geq 1$, solve the equation

$$\frac{F-G_k^n}{p^{2^{k-1}}} = C_k\cdot n\cdot G_k^{n-1} \pmod{p^{2^{k-1}}}$$

for $C_k(x)$ in $Z_{p^{2^{k-1}}}[x]$, and get the next approximation by $G_{k+1} = G_k + p^{2^{k-1}} C_k$.

Note that the price we pay for quadratic convergence is a more difficult equation to solve. By that we mean (1) G_k is different now for each iteration k, (2) the domain $Z_{p^{2^{k-1}}}[x]$ in which the equation has to be solved is growing with k and less well-structured than $Z_p[x]$, and (3) the domain is no longer Euclidean, i.e., division algorithm can not be carried out for arbitrary elements but, fortunately, it is still possible for the particular problem since $G_k \equiv G_1 \pmod p$ for all $k \geq 1$.

3.2 Power Series Applications

The space of polynomials in a particular variable x over a coefficient domain D, D[x], is always contained in the space of power series in x over D, denoted by D{x}, whose elements are of the form $\sum_{i=o}^{\infty} a_i x^i$ with a_i in D. But the role of the variable x, which is certainly an "irreducible" or "prime" element of D{x}, can readily be identified with that of a prime p in the p-adic representation of elements in Z. As a matter of fact, all the definitions and properties of p-adic representation and approximation developed in Section 2 applies identically in D{x} with x replacing p. Thus power series representation is a particular extension of the p-adic representation of integers. In

fact, any unique factorization domain (u.f.d.) has a p-adic representation with respect to a particular "prime" element of that domain. For example, in the u.f.d. Z, any element a can be written in a p-adic form, so that actually $Z = Z_p[[p]]$. Similarly $Z[x] = (Z_p[[p]])[[x]] = (Z_p[[x]])[[p]]$.

Finiteness of the series in p-adic representation plays an essential role. Computations with infinite quantities are generally difficult and sometimes infeasible. Approximations of infinite series by a sequence of finite series is the usual remedy. In effect, we like to identify D{x} with D[[x]] by the argument that an arbitrary power series of D{x} can always be approximated to arbitrary number of terms, m, by an element of D[[x]] via x^m congruence (mod x^m). If we carefully (re-)examine known methods for computation with infinite power series, we will find that, rather than dealing with infinite quantities, we prefer to use elements of D[[x]] together with some iteration scheme in most cases.

Power series reciprocal, division, reversion and composition can all be viewed as computations in D[[x]], where D is assumed to be a field as a rule. Lipson [LIP76] gives a rigorous treatment of the first three topics, attributing certain ideas to Kung [KUN74]. All three operations can be shown to be succeptible to the Newtonian iteration approach. Brent and Kung [B&K76] showed that the fourth operation-power series composition-also has an iterative approach in D[[x]]. We will not repeat their derivations, but only reiterate the importance of Newtonian iteration techniques in power series space, which we identify with D[[x]]. The author would like to acknowledge that J. D. Lipson first brought out the idea of Newton's method for power series inversion during a seminar at IBM Research Center. That furnished the inspiration for this author to extend, generalize, and unify a wide variety of algebraic techniques through the Newtonian point of view, as we shall see in this paper.

3.2 Another Example

Before we leave the topic of single variable mappings Φ, we give another example of power series related application. This example will show the correctness of considering p-adic approximation techniques as well as the possibility of relaxing some requirements for the rigid application of Newton's iteration as compared to those put forth by Lipson.

The Legendre polynomials have been widely used in mathematics and, consequently, have had numerous definitions and derivations for them. One standard definition for Legendre polynomials is

$$P_n(x) = \frac{1}{2^n n!} \frac{d^n}{dx^n} (x^2-1)^n \quad \text{for } n = 0,1,2,\ldots$$

so that, the first few are

$$P_0(x) = 1,\ P_1(x) = x,\ P_2(x) = \frac{3x^2-1}{2},$$

$$P_3(x) = \frac{5x^3-3x}{2},\ \cdots$$

Two other commonly used formulations for them are

(1) Recurrence Relation:

$$P_{n+1}(x) = \frac{2(n+1)x\,P_n(x) - n\,P_{n-1}(x)}{n+1} \quad n=1,2,\ldots$$

(2) Generating Function:

$$G(x,t) = (1-2xt+t^2)^{-1/2} = \sum_{n=0}^{\infty} P_n(x)t^n .$$

It is the last formulation of Legendre Polynomials that can easily be converted to a root finding approximation process. First, we note that the domain of concern is (Q[x])[[t]], which is a p-adic space in t so that t plays the same role as any prime p. Define a function

$$\Phi(G) = (1-2xt+t^2)G^2-1 .$$

We can now apply the Newtonian derivation technique on this function to find approximations to a root of $\Phi(G) = 0$ in the domain of p-adic series (Q[x])[[t]). By the first order Taylor series expansion, we need to solve the equation

$$\Phi(G_k) + \Delta G_k \cdot \frac{d\Phi(G_k)}{dG} \equiv 0 \qquad (\text{mod } q_{k+1})$$

$$\text{or } \Phi(G_k)/q_k + (\Delta G_k/q_k)\frac{d\Phi(G_k)}{dG} \equiv 0 \qquad (\text{mod } q_{k+1}/q_k) .$$

Initially $G_1(x,t) = P_0(x) = 1$.

First we choose $q_k = t^k$, $k = 1,2,\ldots$.

The equation to be solved becomes

$$((1-2xt+t^2)G_k^2-1)/t^k + (\Delta G_k/t^k)\frac{d\Phi(G_k)}{dG} \equiv 0 \pmod t$$

So, for k = 1, we have

$$(t^2-2xt)/t \equiv -2(\Delta G_1/t) \pmod t$$

$$\text{or } -2x \equiv -2(\Delta G_1/t) \pmod t$$

$$\text{or } (\Delta G_1/t) \equiv x \pmod t.$$

Thus $G_2(x,t) = G_1(x,t)+\Delta G_1 = 1+xt = P_0(x) + P_1(x)t$.

I.e., $P_0(x) = 1$, $P_1(x) = x$.

One more step takes us to $P_2(x)$, with k = 2: solve

$$[x^2t^4-2(x^3-x)t^3-(3x^2-1)t^2]/t^2$$

$$\equiv [-2xt^3+(4x^2-2)t^2+2xt-2](\Delta G_2/t^2) \pmod t$$

$$\text{or } -3x^2+1 \equiv -2(\Delta G_2/t^2) \pmod t$$

$$\text{so } (\Delta G_2/t^2) \equiv \frac{3x^2-1}{2} \pmod t$$

$$\text{and } G_3(x,t) = 1+xt+\frac{3x^2-1}{2}t^2, \text{ i.e., } P_2(x) = \frac{3x^2-1}{2}.$$

Obviously this process can be carried on (with computational improvements) to get better and better

approximations to the root G of Φ while getting more and more Legendre polynomials $P_k(x)$.

Next, we choose $q_k = t^{2^{k-1}}$, for comparison purposes. The equation to solve for each k becomes

$$(1-(1-2xt+t^2)G_k^2)/t^{2^{k-1}}$$

$$\equiv (\Delta G_k/t^{2^{k-1}}) \frac{d\Phi(G_k)}{dG} \pmod{t^{2^{k-1}}}.$$

For k = 1, the computation is same as the previous one, so that $G_2(x,t) = 1+xt$. The iteration for k = 2 is, however, different. The equation to be solved now is

$$(-2x^3+2x)t-3x^2+1 \equiv (-2xt+2)(\Delta G_2/t^2) \pmod{t^2}$$

It is necessary to first compute $(-2xt+2)^{-1}$ in $(Q[x])[[t]]$ modulo t^2, which is found to be $\frac{xt+1}{2}$.

Thus $\Delta G_2/t^2 \equiv \frac{5x^3-3x}{2} t + \frac{3x^2-1}{2}$ and

$$G_3(x,t) = G_2+\Delta G_2 = 1+xt+ \frac{3x^2-1}{2} t^2 + \frac{5x^3-3x}{2} t^3,$$

i.e., we have found $P_2(x) = \frac{3x^2-1}{2}$ and

$P_3(x) = \frac{5x^3-3x}{2}$.

Note that two more Legendre polynomials were found in this step at the cost of computing a series inverse or division of series. Clearly, there is an issue of trade-off in computational efficiency, which we will address briefly later.

4.0 Coupled p-adic Construction

In the last section we derived some and briefly mentioned other possible applications of p-adic construction via the Newtonian approach to various computations with polynomials or power series. One peculiar fact, which an observant reader will recognize, is that the function Φ has only one argument, so that a zero of Φ is a single element from the domain of concern. Some of the results appear in [LIP76] and [B&K76] even though they restricted their considerations mainly to the space of (infinite) power series over fields (the rational numbers in general). As we shall see, the Newtonian approach is much more general and powerful. In fact, its major impact in algebraic computations does not lie in dealing with infinite series over fields, which are fields themselves, or any truncated series thereof, but in making available computational schemes for finite polynomials over u.f.d's, which are u.f.d's themselves. Rings of polynomials are much more general, though less structured, than fields of infinite series. Essentially all quantities that a computational method or machine can effectively deal with are indeed finite.

From this point on, we will consider functions Φ which have two arguments. The Newtonian approach to the derivation of constructive methods is similar but the resulting algorithms and the intermediate computational considerations can be quite different to those related to single argument functions. Since we have realized that $Z[x] = (Z_p[[x]])[[p]] = (Z_p[x])[p]$, we will also use the simpler notation Z[x] whenever we discuss p-adic constructions.

4.1 Newton's Iteration on P-adic Space

Let Φ be a bivariate function on Z[x], $\Phi(G,H):Z[x] \times Z[x] \to Z[x]$. Then the first order bivariate Taylor series expansion of Φ about the point (G_k, H_k) is

$$\Phi(G,H) = \Phi(G_k,H_k) + (G-G_k) \frac{\partial\Phi(G_k,H_k)}{\partial G}$$

$$+ (H-H_k)\frac{\partial\Phi(G_k,H_k)}{\partial H}.$$

Again, if (G,H) is assumed to be a root or sufficiently near a root of Φ, then $\Phi(G,H)$ can be considered zero and we have

$$\Phi(G_k,H_k) = -(G-G_k) \frac{\partial\Phi(G_k,H_k)}{\partial G}$$

$$-(H-H_k) \frac{\partial\Phi(G_k,H_k)}{\partial H}.$$

Now consider a particular function Φ to be $\Phi(G,H) = F - GH$ for some given F in Z[x]. (Note that the root of this function Φ, i.e., (G,H), will clearly be factors of the polynomial F.) Then, the above equation becomes

$$F-G_kH_k = \Phi(G_k,H_k) = (G-G_k) \cdot H_k + (H-H_k) \cdot G_k.$$

Since there are two unknowns, G and H, in this case, we do not solve for them as in the derivation of Newton's formula, but we consider the differences as "updates",

$$\Delta G_k = G-G_k, \ \Delta H_k = H-H_k,$$

and we have the equation

$$F - G_kH_k = G_k \cdot \Delta H_k + H_k \cdot \Delta G_k \qquad (1)$$

to solve at each iteration. The solutions for each equation are then used to update the current approximations to the desired zero of Φ by

$$G_{k+1} = G_k + \Delta G_k \text{ and } H_{k+1} = H_k + \Delta H_k. \qquad (2)$$

The convergence of Newton's method is guaranteed by a "good initial guess". Here we let (G_1,H_1) satisfying $\Phi(G_1,H_1) = 0 \pmod p$ (i.e., $F \equiv G_1H_1 \pmod p$) be the given "initial guess" which is "good" in the sense that it approximates F(G,H) = 0. Note that G_1 and H_1 are

in $Z_p[x]$ which is contained in $Z[x]$, so that the approximation procedure is carried out in the proper domain of Φ. Inductively, on the kth step, assume the point (G_k,H_k) has been found such that $\Phi(G_k,H_k)$ approximates zero in the sense that

$$\Phi(G_k,H_k) = F - G_kH_k \equiv 0 \pmod{q_k} \qquad (3)$$

Here q_k denotes some power of p which will be specified later. The previous derivation indicates that the equation

$$G_k \cdot \Delta H_k + H_k \cdot \Delta G_k = F - G_kH_k$$

must be solved for the next approximation (G_{k+1},H_{k+1}) satisfying

$$\Phi(G_{k+1},H_{k+1}) = F - G_{k+1}H_{k+1} \equiv 0 \pmod{q_{k+1}} \qquad (4)$$

for some higher power of p, q_{k+1}.

We observe that it would certainly be wasteful to solve the equation for the updates, G_k and H_k, to its full accuracy in the entire domain $Z[x]$ while the next approximation G_{k+1} and H_{k+1} are expected to be accurate only modulo q_{k+1} or in $Z_{q_{k+1}}[x]$. Thus it is sufficient to solve the equation in $Z_{q_{k+1}}[x]$:

$$G_k \cdot \Delta H_k + H_k \cdot \Delta G_k = F - G_kH_k \pmod{q_{k+1}} \qquad (5)$$

At this point, another observation can be made so that the equation can be solved even in a simpler domain, $Z_{(q_{k+1}/q_k)}[x]$. This improvement is achieved (due to (3) which implies q_k divides $F-G_kH_k$) by replacing (5) by

$$G_kA_k + H_kB_k = (F-G_kH_k)/q_k \pmod{q_{k+1}/q_k} \qquad (6)$$

to solve for A_k and B_k in $Z_{(q_{k+1}/q_k)}[x]$ and replace (2) by updating with current approximation $G_{k+1} = G_k + q_kB_k$ and $H_{k+1} = H_k + q_kA_k$. In effect, we have replaced ΔG_k and ΔH_k by q_kB_k and q_kA_k respectively. This is justified because G_{k+1} and H_{k+1} are to be higher order p-adic approximations, i.e., q_k must divide the relative errors -- $G_{k+1}-G_k$ and $H_{k+1}-H_k$. It can be proved that the solvability of this equation (6) depends only on the condition that G_1 and H_1 are relatively prime, which we do assume.

We will now specify q_k in two different ways to get the two desired algorithms. These choices of q_k, however, are by no means the only possible or interesting ones.

4.2 Zassenhaus' Construction

Let $q_k = p^{2^{k-1}}$ for $k = 1,2,\ldots$. Then by (6) the equation to solve at each iteration is

$$G_kA_k + H_kB_k = (F-G_kH_k)/p^{2^{k-1}} \pmod{p^{2^{k-1}}} \qquad (7)$$

with the updating formula being

$$G_{k+1} = G_k + p^{2^{k-1}}B_k \text{ and } H_{k+1} = H_k + p^{2^{k-1}}A_k \qquad (8)$$

Expressions (7) and (8) are precisely the iteration formulas that accomplish the desired results of Zassenhaus' Construction stated in Section 2 of this paper. A p-adic series approximation is achieved such that for any $k \geq 1$

$$G_{k+1} = G_1 + pB_1 + p^2B_2 + p^4B_3 + p^8B_4 + \ldots + p^{2^{k-1}}B_k$$

$$H_{k+1} = H_1 + pA_1 + p^2A_2 + p^4A_3 + p^8A_4 + \ldots + p^{2^{k-1}}A_k .$$

Since G_k, H_k, A_k and B_k are all in $z_{q_k}[x]$ which is the space of p-adic series with $\log_p q_k = 2^{k-1}$ terms, each updating step (8) doubles the number of terms in the current p-adic approximation (G_k,H_k) of the desired root of $\Phi(G,H)$. Hence the Zassenhaus' Construction exhibits quadratic convergence.

4.3 Hensel Lemma

On the other hand, let $q_k = p^k$ for $k = 1,2,\ldots$, then, according to (6), the equation to be solved is

$$G_kA_k + H_kB_k = (F-G_kH_k)/p^k \pmod{p}.$$

But modulo p, $G_k \equiv G_1$ and $H_k \equiv H_1$ for all k. Thus, solving the equation

$$G_1A_k + H_1B_k = (F-G_kH_k)/p^k \pmod{p} \qquad (9)$$

for A_k and B_k in $Z_p[x]$ is the "inner loop" of the iterative procedure. The following is then used for updating

$$G_{k+1} = G_k + p^kB_k \text{ and } H_{k+1} = H_k + p^kA_k. \qquad (10)$$

This derivation precisely established the validity and an inductive construction of Hensel's Lemma. This construction also constitutes a p-adic series approximation with

$$G_{k+1} = G_1 + pB_1 + p^2B_2 + p^3B_3 + \ldots + p^kB_k$$

$$H_{k+1} = H_1 + pA_1 + p^2A_2 + p^3A_3 + \ldots + p^kA_k$$

for any $n \geq 1$.

Since A_k and B_k are in $Z_p[x]$ which is the space of p-adic series with one term each, the updating step (10) clearly adds one more term to the current p-adic approximation (G_k, H_k) each time. Hence Hensel's Construction is only a linearly convergent approximation method.

4.4 Order of Convergence Versus Computational Efficiency

Summarizing, we began by considering a Newtonian approximation method on the space of p-adic series isomorphic to Z[x]. For a particular function $\phi(G, H) = F-GH$ whose roots constitute factors of F, we derived both Zassenhaus' and Hensel's Constructions by choosing a certain sequence $\{q_k\}$ of powers of p. The two choices for $\{q_k\}$ result in two p-adic approximation methods that exhibit different convergence behavior, one quadratic and the other linear.

Even though these are not the only two interesting cases, an important question to pose is which method achieves better computational efficiency (e.g., in terms of the cost or number of single precision integer multiplications). For numerical computations the quadratically convergent Newtonian iterations are usually more efficient than linear methods (c.f. [TRA64]. As a result, the Zassenhaus' construction was used extensively for factoring integral polynomials even for multivariate polynomials [MUS71] and [W&R75].

Miola and Yun [M&Y74] made a detailed study and analysis of the algebraic algorithms of Hensel and Zassenhaus in 1974. Their finding, both from theoretical analysis of the algorithms, as presented with only obvious computational improvements, and experimental data, showed that the cost for Zassenhaus' construction is higher than that of Hensel's construction for achieving the same accuracy in the p-adic approximation of the results, except for the first step of the constructions (from modulo p to modulo p^2) which is identical for both methods. Although this result is somewhat surprising (and rather lengthy to derive), the basic reasons are not difficult to explain. From our Newtonian point of view, equations such as (1), (5), (6), (7), and (9) are the so-called "inner loop" of each Newtonian iteration, and most of the computational effort has to be put here, since the "outer loop" of a Newtonian iteration such as (2), (8), and (10) usually involves only simple updating (e.g., adding). For most numerical Newtonian iteration procedures, the inner loops usually consist of function and derivative evaluations. But the solution of Diophantine polynomial equations is the main task to be performed in the inner loops of these algebraic constructions. Zassenhaus' construction requires solving (7) modulo $p^{2^{k-1}}$ for $k = 1,2,\ldots,$ while Hensel's construction only needs to solve (9) in $Z_p[x]$ with fixed G_1 and H_1 for each $k \geq 1$. Since $Z_p[x]$ is a smaller and more structured domain (Z_p is a field) and repeated use of G_1 and H_1 makes it possible to use some "preconditioning", these constitute the main reasons for Hensel's construction to be more efficient. (One preconditioning is to solve $G_1a_i + H_1b_i = x^i$ for $i = 0,1,\ldots,d$ where $d \leq$ degree of F (= degree of G_1 + degree of H_1). Then any solution A_k and B_k for a particular right-hand-side of (9) can be expressed as scalar linear combinations of the a_i's and b_i's respectively). Thus we encounter a set of Newtonian iteration procedures where the quadratically convergent method turns out to be computationally more costly (i.e., less efficient) than the linearly convergent construction, because the cost of inner loop for each iteration is too high. This is another case among many others in symbolic and algebraic computations where simple-minded intuition from numerical algorithmic studies can lead us astray ([BRO71] [COL66] [FAT74] [G&J73]).

4.5 Applications of Coupled P-adic Constructions

Derivations or comparisons of constructive methods deserve only academic interest unless the techniques have actual computational utilizations. Both Hensel's and Zassenhaus' constructions, derived via Newtonian point of view in this section, have found, or in fact initiated, numerous applications in algebraic computations. We will list and briefly discuss each, giving only motivational definitions while referring the reader to appropriate sources for further details.

(1) Factorization of Polynomials:

$F = G \cdot H$, F given and G, H unknown factors.

As we mentioned in Section 2, p-adic representation and approximations were the original underlying ideas for lifting factors modulo p to larger domains over fifty years ago. Hensel provided the foundation [HEN13] and, more than half a century later, Zassenhaus gave the added incentive of quadratic convergence [ZAS69]. Utilizing these lifting techniques, Musser [MUS71] and Wang and Rothschild [W&R75] devised and implemented methods for factoring polynomials over integers or rational numbers, which are deterministic, constructive, and much more efficient than before (e.g., Kronecker's method [MUS71]). The appropriateness of these p-adic approximation techniques to factorization is unquestionable. When the factors are lifted to a modulus p^k that is larger than any integral coefficient, the correspondence between the factors modulo p^k and the true factors over Z becomes quite clear. Yun [YUN74a] pointed out that the mod-p^k factors can differ from the true factors by a multiple of a unit in Z_{p^k} (barring the existence of extraneous factors). The idea of "forcing-in" the leading coefficients was proposed, tested, and shown to be especially efficient in the multivariate case by Yun. (Subsequently, it became known that Collins gave a lecture in 1969 discussing a similar idea, even though the computational merit was not fully recognized at that time.) Musser adopted this idea in his later version of factorization algorithm [MUS75].

(2) Polynomial Division with (or without) Remainders:

Given polynomials F and S, the problem is to find quotient Q and remainder R such that

$$F = QS + R$$

The division problem can obviously be put into the linear Diophantine equation form of $F = QS + RT$ with $T = 1$. In non-Euclidean domains such as $Z[x]$, pseudo-division is necessary. But the multiplier can be predicted to always divide $\ell c(S)^{\deg(F)-\deg(S)+1}$, so that if we simply multiply F by this known element of the coefficient domain then the p-adic construction can be carried out smoothly. Yun [YUN74b] gave a more detailed algorithm for doing just this p-adic polynomial division, together with some examples.

(3) Partial Fraction Decomposition:

Given a rational function with factored denominator $\frac{F}{S \cdot T}$, getting its partial fraction decomposition means to compute numerator polynomials G and H such that $\frac{F}{S \cdot T} = \frac{H}{S} + \frac{G}{T}$.

Again the correspondence of this problem to solving polynomial Diophantine equations is quite clear. If solutions G and H are found so that $F = SG + TH$, then $\frac{F}{S \cdot T} = \frac{SG+TH}{S \cdot T} = \frac{G}{T} + \frac{H}{S}$ the desired partial fraction decomposition. A word of caution is needed here. As opposed to the case of division, the correct multiplier to F (in fact, it is the resultant of S and T) necessary for p-adic approximation to go through in the same domain which is non-Euclidean, is either too large or too costly to compute. In other words, the multiplier is effectively unknown. Thus the p-adic approximation procedure applies in this case only in Euclidean domains (which is guaranteed if the coefficient domain is a field).

(4) Systems of Linear Diophantine Equations over Polynomials:

$(A_{ij})(B_j) = (C_i)$ with A_{ij} and C_j given;

$i,j = 1,\ldots,n$.

This is a direct generalization of solving one Diophantine equation. The difference is only that one equation in two unknowns results in a parametrized family of solutions (even though any solution can be uniquely determined by putting constraints on the degrees) but a system of $n \times n$ linear equations has a unique solution even in polynomial domains (if a solution exists). The author is currently working on this problem with the goal of solving SIGSAM Problem #10 posed by Loos in full generality. A set of unpublished lecture notes and solved sample problems is available upon request by interested readers.

6.0 Generalizations, Computational Costs, and Conclusion

6.1 The Multivariate Cases

Throughout our previous discussions, we have concentrated mainly on p-adic representations and approximation procedures in $Z[x]$ or $Z[[x]]$. We have been, however, using carefully chosen terminology for intended generalizations of this section. We used, for instance, "u.f.d." or "Euclidean domain" to describe $Z[x]$ or $Z_p[x]$, since a general polynomial space over an arbitrary ring is not always Euclidean. Properties of a Euclidean domain are especially important for p-adic computations, as we saw in power series related methods. On the other hand, simple u.f.d. properties suffice to carry out p-adic procedures for many other problems, such as factorization, GCD computation, and division of polynomials. Unfortunately, the space of multivariate polynomials over Z or Q (the rational numbers), which is our main direction of generalization, is always a u.f.d. but not necessarily a Euclidean domain.

Consider the space of bivariate polynomials over Z, $Z[x,y]$. Any $F(x,y)$ in $Z[x,y]$ can be written as $\sum_{i=o}^{u} \left(\sum_{j=o}^{v} \left(\sum_{k=o}^{w} F_{ijk} p^k \right) y^j \right) x^i$ for some F_{ijk} in Z_p w.r.t. a chosen prime p in Z.

Equivalently, $F(x,y) = \sum_{k=o}^{w} \left(\sum_{j=o}^{v} F_{jk}(x) y^j \right) p^k$ for $F_{jk}(x)$ in $Z_p[x]$. This is a generalization to the p-adic representation introduced earlier. Here we see that y plays the same role as p and it is clearly a prime in the coefficient space with respect to the main variable x. Thus any prime or irreducible polynomial in y can be used for representation purposes. In particular, we often choose linear monic ones such as y-b with b taken from Z_p. Thus, polynomials modulo y-b is equivalent to evaluating it at the point $y = b$. The <u>generalized p-adic representation</u> of a multivariate polynomial $F(x,y_1,\ldots,y_v)$ in $Z_p[x,y_1,\ldots,y_v]$ w.r.t. a chosen set of integral value $\{b_1,\ldots,b_v\}$ is

$$F(x,y_1,\ldots,y_v) = F(x,b_1,\ldots,b_v)$$

$$+ \sum_{i=1}^{v} F_i(x)(y_i-b_i)$$

$$+ \sum_{i=1}^{v} \sum_{j=1}^{v} F_{ij}(x)(y_i-b_i)(y_j-b_j)$$

$$+ \sum_{i=1}^{v} \sum_{j=1}^{v} \sum_{k=1}^{v} F_{ijk}(x)(y_i-b_i)(y_j-b_j)(y_k-b_k)$$

$$+ \ldots$$

(2) Polynomial Greatest Common Divisor (GCD):

Given polynomials E and F, if $G_1 = \gcd(E,F) \pmod p$ and $H_1 = F/G_1 \pmod p$ where $\gcd(G_1,H_1) = 1$, then obviously the same p-adic constructions can be used to lift G_1 to G_k and p^k and, finally, to a corresponding GCD of E and F over Z.

The usage of p-adic approximation ideas is so similar to that of factorization, but the gain of computational efficiency over the polynomial remainder sequence (PRS) or modular approaches still appeared surprising when the EZGCD algorithm was announced [M&Y73].

(3) Contents of Polynomials or Simultaneous GCD Computation:

Cont(f(x)) = GCD of all polynomial coefficients w.r.t. x.

Observing that the computation of gcd(E,F) needs only apply the p-adic construction on either E or F, we recognize that GCD of more than two polynomials requires not much more effort. Classically $\gcd(F_1,F_2,\ldots,F_m)$ is computed recursively by groups of two:

$$\gcd(\gcd(\ldots\gcd((F_1,F_2),F_3)\ldots),F_m).$$

Simultaneous GCD's of this form can now be computed in a semi-parallel fashion -- by first computing the modulo-p GCD of $F_1,\ldots,F_m$ and then lifting using only one of these F's[YUN74a].

(4) Squarefree Decomposition of Polynomials:

Any primitive polynomial F(x) has a canonical decomposition of the form $F_1^1F_2^2F_3^3\ldots F_k^k$ where $\gcd(F_i,dF_i/dx) = 1$, $\gcd(F_i,F_j) = 1$ for all $i \neq j \leq k$, and $F_k \neq 1$.

Horowitz [HOR71] and Musser [MUS71] gave two closely related algorithms for computing the squarefree decomposition of polynomials by successively finding GCD's. Yun [YUN74a] presented the EZSQFR algorithm which utilizes the decoupled p-adic construction in a more satisfactory way than any of the previously mentioned three. Recently, a new but surprisingly simple method was found [YUN76] which is a clear improvement over Horowitz's and Musser's versions. When this new method is put into the p-adic approximation framework, it appears to be even more superior than the EZSQFR algorithm.

5.0 Decoupled P-adic Constructions

In our derivation of p-adic approximation methods starting from the bivariate Taylor series expansion of Φ in the last section, we have chosen a particular function, $\Phi(G,H) = F - GH$, in order to derive methods for lifting factors of polynomials, The computational procedure so derived is called "coupled p-adic construction", since the unknowns are coupled together through multiplication. We have also briefly mentioned the computational difficulties and remedies related to this coupling (Section 4.5). Since the Newtonian iteration technique is general and the function Φ of the last section is particular, the natural question to ask at this point is whether there is a corresponding p-adic procedure that is "decoupled". The natural answer to this natural question lies in the function

$$\Phi(G,H) = F - SG + TH \quad \text{for given } F,S,T \text{ in } Z[[x]].$$

Now we will be very brief with the derivation, but point out that the name "decoupled p-adic construction" comes from the fact that the unknown, G and H, appear linearly and not coupled through multiplication. Much of the computational difficulties due to coupling, therefore, disappear in this case.

Considering the bivariate function Φ on Z[x]:

$$\Sigma(G,H): Z[x] \times Z[x] \to Z[x] \text{ with}$$

$\Phi(G,h) = F - SG + TH$, the Taylor series derivation gives $F - SG_k - TH_k = S(G-G_k) + T(H-H_k)$. Let $\Delta G_k = G - G_k$ and $\Delta H_k = H - H_k$, then the equation to be solved at each step of the approximation is

$$S\Delta G_k + T\Delta H_k = T - SG_k - TH_k \pmod{q_{k+1}}.$$

Dividing through again by q_k (due to the inductive hypothesis: $F - SG_k - TH_k \equiv 0 \pmod{q_k}$), we have the iteration formula of:

For each $k \geq 1$, solve

$$SA_k + TB_k = (F-SG_k + TH_k)/q_k \pmod{q_{k+1}/q_k}$$

and use the solutions A_k, B_k in $Z_{q_{k+1}/q_k}[x]$ to update the approximation $G_{k+1} = G_k + q_kA_k$ and $H_{k+1} = H_k + q_kB_k$.

5.1 Applications of Decoupled P-adic Construction

(1) Linear Diophantine Equations over Polynomials:

$SG + TH = F$ with given polynomials F,S,T and unknowns G and H.

Obviously, the root of the above function $\Phi(G,H)$ solves this equation. Again the finiteness of the given and unknown polynomials makes a terminating p-adic approximation procedure possible. The "initial guess", G_1 and H_1 such that $SG_1 + TH_1 = F \pmod p$, for this Newtonian iteration plays an important role by guaranteeing the ultimate convergence of the construction. If at any point in the iterative procedure, the equation in $Z_{q_{k+1}/q_k}[x]$ cannot be solved, then the Diophantine equation over Z[x] has no solution.

where the coefficient polynomials in x are from $Z_p[x]$. For $F(x,y_1,\ldots,y_v)$ in $Z[x,y_1,\ldots,y_v]$, each integral coefficient will then be put in a p-adic representation w.r.t. a chosen prime p and the generalized representation will have one more level of summation w.r.t. powers of p while the coefficient polynomials in x are still taken from $Z_p[x]$. This representation provides the basis for multivariate p-adic approximation of polynomials. It was first used by Wang and Rothschild [W&R75] for applying the generalized Hensel's Lemma on factorization problems. Yun [YUN74a] formalized this representation and the related p-adic approximation algorithm. Musser [MUS71] [MUS75] used a somewhat different representation which corresponds more directly to a recursive representation w.r.t. the different (y_i-b_i)'s. Since he wanted to use the quadratic Zassenhaus' Construction for lifting factors, he was required to perform an additional level of lifting, i.e., within one step of his coupled p-adic construction for lifting factors, he is required to apply one step of the decoupled p-adic construction. Thus an implicitly non-Euclidean domain p-adic construction is involved. Even if there is no problem in doing so, the computation is recursive, hence quite costly. Although some studies on the trade-off between the two similar p-adic factor lifting methods (especially in multivariate cases) have been done by Miola and Yun [M&Y74], more careful and thorough analysis is still needed. In any case, similar p-adic approximation methods for multivariate polynomials can be derived via the Newtonian point of view. One should only be careful with guaranteeing that the equations in the Newtonian iteration can be solved in the domain of concern whenever a true solution to the problem actually exists.

6.2 Computing Costs

To give any detailed complete cost analysis of all these computational algorithms is simply not possible in this paper. We can, therefore, only sketch ideas and quote known results. So the following is, by necessity, survey in nature but not in completeness.

Let M(n) be the computing cost in terms of coefficient arithmetic operations (mainly multiplication) for multiplying two polynomials of degree n. By the classical polynomial multiplication algorithm, $M(n) = n^2$, whereas by use of FFT, $M(n) = n \log n$. Lipson [LIP76] gives a convincing argument as to why the cost of Newtonian iteration is dominated by the cost of the last step. Based on that, it is easy to show: cost of division or inversion of power series to n terms = O(M(n)); and, cost of computing r^{th} root of polynomials to n terms = $O(\log r \cdot M(n))$. Elaborating on the Newtonian iteration further, we observe that besides the computation of next update, there is substantial cost involved in function substitution or composition (i.e., computing $\Phi(G_k)$). Lipson simply used the Horner's rule for substitution, which for $\Phi(G) = \sum_{i=o}^{r} \Phi_i G^i$ results in O(rM(n)) as the total asymptotic cost for Newtonian iteration. Brent and Kung [B&K76], using a double layer Newtonian approach (i.e., recursively apply Newton's method also on function composition), were able to reduce the cost to $O(r \cdot \log r \cdot M(n))$. For example, they were able to give an asymptotically fast algorithm for the reversion of power series to n terms in $O((n \log n)^{3/2})$. Using the similar double layer Newtonian approach, we can also establish that the computation of first n Legendre polynomials requires $O((n \log n)^{3/2})$ arithmetic operations in Q[x].

Cost analyses of bivariate Newtonian procedures have not been as systematic, partly because the coupled or decoupled p-adic constructions have proved more efficient and advantageous in multivariate sparse polynomial computations. These problems require a different set of measures for computing cost, and asymptotic analysis makes little sense here. Yun [YUN74a] made a first attempt at establishing some computing cost formula for the multivariate coupled p-adic method. It is too complex to quote out of context, but brings out the need for a more comprehensive analysis and a more comprehensible cost formula, if possible. It can be observed, and Yun did so in [YUN74a], that the cost of polynomial factorization and GCD computation is usually dominated by the cost of p-adic construction. Because of the semi-parallel evaluation of simultaneous GCD's, the cost of computing contents of polynomials is undoubtedly improved (to that of the cost for one p-adic construction) [YUN74a]. The cost of computing squarefree decomposition of polynomials has been established to be the same as that of computing the GCD of the given polynomial and its derivative [YUN76], hence also the same as one p-adic construction.

Moenck [MOE73] showed that performing the extended Euclidean algorithm in a Euclidean domain can be improved (using Strassen's devices) to an asymptotic cost of $O(\log n \cdot M(n))$ where n bounds the degrees of two given polynomials. Solving a linear Diophantine equation over polynomials in the form $SG + TH = F$ with F,S,T given such that $\deg(F) < \deg(G) + \deg(H)$, amounts to extending the results of the Extended Euclidean algorithm, $AS + BT = \gcd(S,T)$, by computing $D = F/\gcd(S,T)$, two multiplications (A·D and B·D), and one division ($AD = QT + G$). The unique solution -- G and $H = Q \cdot S + BD$ -- satisfies $\deg(G) < \deg(T)$ and $\deg(H) \leq \deg(S)$. The cost for getting the solution G and H is, therefore, dominated by that of Extended Euclidean algorithm, hence $O(\log n \cdot M(n))$ where $n \geq \deg(S) \geq \deg(T)$.

Correspondingly, division of polynomials from the p-adic construction point of view is bounded by a cost of $O(\log n \cdot M(n))$. But it has been established [AHU74] that cost of division = O(M(n)). The discrepancy can be explained away by noting the p-adic division is a very special case of solving Diophantine equations, i.e., T = 1. For this special situation the log n factor drops out and O(M(n)) being the cost for division can be established through the p-adic construction viewpoint also.

Similarly, the p-adic construction approach shows that the problem of partial fraction

decomposition can be done at a cost of $O(\log n \cdot M(n))$. Kung and Tong [K&T76] were able to establish the same cost bound independently.

6.3 Conclusion

This paper has, hopefully, brought out the generality of the Newtonian approach and the importance of finite (p-adic) approximation in algebraic computations. One key observation in the method for derivation is the fact that the first order Taylor series expansion of the function, we wish to find the root of, is the starting point and, in some sense, the finish. That is, we derive the equation to be solved for the update from it rather than explicitly displaying the update, $-\Phi(x_k)/\Phi'(x_k)$, as the classical Newton's method does. Setting up such an equation to solve in the so-called "inner loop" of Newton's method has the advantage of allowing truncation, i.e., modulo q_{k+1}. This, in turn, leads to solving a simpler equation in a smaller and more well-structured domain, such as $Z_{q_{k+1}/q_k}[x]$. We have actually seen several such simplifications above. In some cases q_{k+1}/q_k reduces to p so that the equation to be solved at each step becomes independent of k as $Z_p[x]$ becomes the basic underlying domain. That is the crucial observation which allows "preconditioning" to be done in $Z_p[x]$ and boosts the computational efficiency of the linearly convergent Hensel-type constructions [M&Y74].

The generality and power of the Newtonian approach on p-adic spaces also come from the generality of the function Φ. In this paper, we have particularized Φ to such forms as $F - G^n$, $(1-2xt+t^2)G^2-1$, F-GH, and F-SG+TH. These choices have produced numerous useful constructive methods, as we have seen. Obviously these are not all the forms that Φ can take. In fact, for the reversion of the power series F(t), for instance, we can let $\Phi(G) = F(G) - t$. (Finite approximation and function composition again play important roles in solving for the root G.) Therefore, we certainly have a fruitful grounds awaiting further cultivation.

We cannot over-emphasize the generality of these algebraic constructions, especially when they are viewed, understood, and unified with such classical analytic concept and technique as Newton's method. Hopefully, this unified point of view sheds more light on the underlying structures, the generalities and restrictions, as well as the advantages and drawbacks of different (algebraic, analytic, or numerical) algorithms. Such a unification can be useful not only for developing symbolic systems, devising new computational methods, and analyzing algebraic algorithms but also in the study of algebraic computational complexity as a whole which seems to lag behind those results in numerical or discrete algorithms.

References

[AHU74] Aho, A. V., Hopcroft, J. E., and Ullman, J. D., The Design and Analysis of Computer Algorithms, Addison-Wedley, 1974.

[BER67] Berlekamp, E. R., "Factoring Polynomials over Finite Field", Bell System Technical Journal, Vol. 46, 1967, pp. 1853-1859.

[BRO71] Brown, W. S., "On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors", JACM, Vol. 18, No. 4, October 1971, pp. 478-504.

[B&K76] Brent, R. P., and Kung, H. T., "Fast Algorithms for Manipulating Formal Power Series", Report, Dept. of Computer Science, Carnegie-Mellon Univ., Jan. 1976.

[COL66] Collins, G. E., "Polynomial Remainder Sequences and Determinants", Am. Math. Monthly, 73, Aug.-Sept. 1966, pp. 708-712.

[FAT74] Fateman, R. J., "On the Computation of Powers of Sparse Polynomials", Studies in Applied Mathematics, Vol. 53, No. 2, June 1974, pp. 145-155.

[GJY75] Griesmer, J. H., Jenks, R. D., and Yun, D. Y. Y., "SCRATCHPAD User's Manual", IBM, T. J. Watson Research Center Report - RA 70, June 1975.

[G&J73] Gentleman, W. M. and Johnson, S. C., "Analysis of Algorithms, A Case Study: Determinants of Polynomials", Proceedings of 5th Annual ACM Symposium on the Theory of Computing, Austin, Texas (1973). pp. 135-142.

[HEN13] Hensel, K., Zahlentheorie, Goschen, Berlin and Leipzig, 1913.

[HOR71] Horowitz, E., "Algorithms for Partial Fraction Decomposition and Rational Function Integration", [PET71], pp. 441-457.

[KNU69] Knuth, D. E., The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, Addison-Wesley, Reading, Mass., 1969.

[KUN74] Kung, H. T., "On Computing Reciprocals of Power Series", Numer. Math., 22, pp. 341-348, Springer-Verlag, 1974.

[K&T76] Kung, H. T., and Tong, D. M., "Fast Algorithm for Partial Fraction Decomposition", Report, Dept. of C. S., Carnegie-Mellon Univ., Jan. 1976.

[LIP76] Lipson, J. D., "Newton's Method: A Great Algebraic Algorithm", These Proceedings, Aug. 1976.

[MAC73] MACSYMA User's Manual, by Bogen, R., et al., Project MAC, M.I.T., Cambridge, Mass., 1973.

[MOE73] Moenck, R., "Fast Computation of GCD's", Proceedings of 5th Annual ACM Symposium on Theory of Computing, pp. 142-151.

[MUS71] Musser, D. R., "Algorithms for Polynomial Factorization", Ph.D. Thesis, C. S. Dept. Univ. of Wisconsin, Aug. 1971.

[MUS75] Musser, D. R., "Multivariate Polynomial Factorization", JACM, Vol. 22, No. 2, April 1975, pp. 291-307.

[M&Y73] Moses, J. and Yun, D. Y. Y., "The EZ GCD Algorithm", Proc. of ACM Annual Conference, Aug. 1973, Atlanta, pp. 159-166.

[M&Y74] Miola, A. and Yun, D. Y. Y., "The Computational Aspects of Hensel-type Univariate Polynomial Greatest Common Divisor Algorithms", Proc. EUROSAM '74, Aug. 1974, pp. 46-54.

[PET71] Proc. of the Second Symposium on Symbolic and Algebraic Maniuplation, Petrick, S.R., ed., ACM, March 1971.

[TRA64] Traub, J. F., Iterative Methods for the Solution of Equations, Prentice-Hall Inc., 1964.

[W&R75] Wang, P. S. and Rothschild, L. P., "Factoring Multivariate Polynomials over the Integers", Math. of Comp. Vol. 29, No. 131, July 1975, pp. 935-950.

[YUN74a] Yun, D. Y. Y., "The Hensel Lemma in Algebraic Manipulation", Ph.D. Thesis, Dept. of Math. MAC-TR-138, M.I.T., Nov. 1974.

[YUN74b] Yun, D. Y. Y., "A p-adic Division with Remainder Algorithm", ACM SIGSAM Bulletin, Vol. 8, No. 4, November 1974 (Issue No. 32), pp. 27-32.

[YUN75] Yun, D. Y. Y., "Nensel Meets Newton -- Algebraic Construction in an Analytic Setting", Analytic Computational Complexity, Proc. of CMU Symposium, Traub, J., ed., Academic Press, April 1975.

[YUN76] Yun, D. Y. Y., "On Squarefree Decomposition Algorithms", Proc. of 1976 ACM Symposium on Symbolic and Algebraic Computation, Aug. 1976.

[ZAS69] Zassenhaus, H., "On Hensel Factorization I", Journal of Number Theory, Vol. 1, 1969, pp. 291-311.

Newton's Method: A Great Algebraic Algorithm †

John D. Lipson
Department of Computer Science
University of Toronto

ABSTRACT

The analytic concepts of approximation, convergence, differentiation, and Taylor series expansion are applied and interpreted in the context of an abstract power series domain. Newton's method is then shown to be applicable to solving for a power series root of a polynomial with power series coefficients, resulting in fast algorithms for a variety of power series manipulation problems. Sample applications of a FORMAC implementation of Newton's method as an algebraic algorithm are presented.

1. INTRODUCTION

1.1. Polynomial root-finding in an algebraic setting

In this paper we investigate the problem of polynomial root-finding in an algebraic, as opposed to the usual numerical, setting.

Let D be an integral domain and D[x] the set of all polynomials $a(x) = \sum_{i=0}^{\infty} a_i x^i$ with coefficients $a_i \in D$ (only finitely many a_i nonzero). As is well known, D[x] is itself an integral domain under ordinary polynomial addition and multiplication.

D might be any of the familiar domains of algebra, such as

$\mathbb{Z}$ the domain of integers,
$\mathbb{Q}$ the rational field,
$\mathbb{R}$ the real field,
$\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$, the field of integers mod p (p a prime).

For us the principal choice of D will be a (formal) power series domain F[[t]], where F[[t]] denotes the set of all power series $a(t) = \sum_{i=0}^{\infty} a_i t^i$ with coefficients $a_i \in F$ (F a field), under power series addition and multiplication.

Root-finding. A root of a polynomial $f(x) \in D[x]$ is an element $\bar{x} \in D$ such that $f(\bar{x}) = 0$. For purposes of unity and comparison we pose our root-finding problem in a form that is sufficiently general to cover both the well-studied numerical case and the algebraic case which is the topic of this paper.

Root-finding problem. Given a polynomial $f(x) \in D[x]$ compute $\hat{x} \in D$ such that $\hat{x}$ is a "good approximation" (in some precise sense) to a root $\bar{x}$ of $f(x)$.

Taking D to be the real or complex field yields the familiar numerical problem of computing a root of a real or complex polynomial. Taking D to be a power series domain F[[t]] yields the essentially algebraic problem of computing a root $\bar{x}(t) = \sum_{i=0}^{\infty} \bar{x}_i t^i$ of a polynomial $f(x) = \sum_{i=0}^{\infty} f_i x^i$ with power series coefficients $f_i = f_i(t) \in F[[t]]$. We regard this case of our general root-finding problem to be an algebraic, rather than a numerical, computational problem, since the field F need not have any analytic structure; for example, F might be a finite field.

The notion of approximation is central to the algebraic case of our polynomial root-finding problem, just as it is for the numerical case.

Example. Consider $f(x) = x^2 - a \in D[x]$. Then a root $\bar{x}$ of $f(x)$ yields a square root of $a \in D$.

† This research was supported in part by the National Research Council of Canada under Grant A-7641.

Case i (numerical). Take $D = \underline{R}$ and $a = 2$. Then $\hat{x} = 1.4142$ yields an approximation to $\sqrt{2}$ which is correct to five figures.

Case ii (algebraic). Take $D = \underline{Q}[[t]]$ and $a(t) = 1 + t + 2t^2 + 3t^3 + \ldots$. Then

$$\hat{x}(t) = 1 + 1/2\ t + 7/8\ t^2 + 17/16\ t^3 + 139/128\ t^4$$

yields an approximation to the power series $\sqrt{a(t)} \in \underline{Q}[[t]]$ that is correct to five terms. Or take $D = \underline{Z}_7[[t]]$ and $a(t) = 1 + t$. Then

$$\hat{x}(t) = 1 + 4t + 6t^2 + 4t^3 + t^4$$

yields an approximation to $\sqrt{1+t} \in \underline{Z}_7[[t]]$ that is correct to five terms.

1.2. Newton's method; summary of results

By Newton's method for finding a root of the equation $f(x) = 0$ we mean the computation of a sequence of iterates $x_1, x_2, x_3, \ldots$ according to the Newton iteration

$$x_{k+1} = x_k - f(x_k)/f'(x_k),$$

starting from some initial approximation x_0 to the desired root $\bar{x}$.

The convergence properties of Newton's method in its time-honoured role as a numerical root-finding algorithm are well-known (e.g., see [6, Sec. 4.7]). If the method converges at all to some root $\bar{x}$, then it converges quadratically, meaning that $\epsilon_{k+1} = C\epsilon_k^2$, where $\epsilon_k = \bar{x} - x_k$ is the error in the k-th iterate, and C is some constant (the so-called asymptotic error constant). For the case of real root-finding, this implies that the number of correct decimal places in an iterate approximately doubles with each successive iteration. The other side of the convergence coin, however, is a bad one. Newton's method is only locally convergent, meaning that convergence to a root $\bar{x}$ is assured only if the initial approximation, x_0, is sufficiently close to $\bar{x}$.

In summary: Newton's method, if it works, works very well.

So much for Newton's method in its familiar numerical setting. The overall objective of this paper is to establish the efficacy of Newton's method in the purely algebraic setting of the polynomial root-finding problem $f(x) = 0$ over a power series domain $F[[t]]$. Our intent is also to compare and contrast the behaviour of Newton's method as an algebraic algorithm with its above-mentioned well-known behaviour as a numerical algorithm.

In Section 2 (mathematical preliminaries) we gather together some ideas from the algebra of formal power series that yield sharp analogues to the analytic concepts of approximation, convergence, and differentiation.

In Section 3 (main results) we apply the ideas of Section 2 to establish Newton's method as an algebraic (formal power series manipulation) algorithm. In particular, we prove an algebraic analogue of quadratic convergence and give a sharp (and useful) global convergence criterion.

In Section 4 (computational aspects) we derive a computing time bound for Newton's method as a polynomial rootfinder over a power series domain. Sample computational results of a FORMAC implementation are given to further demonstrate the practicality of Newton's method as a power series manipulation algorithm.

1.3. An historical perspective

The application of Newton's method to algebraic computing, in contrast to its time-honoured role in numerical computing, is of quite recent vintage. The computational problems that have provided the impetus for this algebraic development of Newton's method have all involved division -- of integers, polynomials, and power series.

The classical division algorithms ("long division") for dividing an n digit integer by an m digit integer and for dividing a degree n polynomial by a degree m polynomial (producing in each case a quotient and remainder) both require time $O(n^2)$ in the worst case $m \approx n/2$. And for the case of dividing the first n terms of a power series by the first n terms of another power series (producing the first n terms of the quotient power series), classical power series division also requires time $O(n^2)$. (Note 1.)

The first speedup of any of these division problems was due to S. Cook [4, pp. 81-86] who applied Newton's method to reduce the time to perform (large) integer division to that of (large) integer multiplication -- thus from $O(n^2)$ to $O(n \log n \log \log n)$ [n = number of digits] using the currently best multiplication algorithm of Schönhage and Strassen [12]. (Actually, Cook applied Newton's method to compute an arbitrarily high precision reciprocal of an integer u. But the integer quotient and remainder that results when an integer v is divided by an integer u are expressible as $q = v \cdot (1/u)$ and $r = u - v \cdot q$, and it is readily shown (taking into account that (1/u) need only be known to finite precision) that Cook's fast reciprocal algorithm does indeed

reduce the time to perform quotient-remainder integer division to that of integer multiplication.)

Of course Newton's method has long been used to compute reciprocals of fixed precision (floating point) numbers. The distinctive feature of Cook's application was its variable precision aspects, wherein the computations of successive iterations involved integers of increasingly higher precision (with the position of the decimal points simply being understood or readily determined). Since all such integer calculations were carried out exactly (i.e., with no rounding errors), we regard Cook's application of Newton's method as being essentially algebraic (or "seminumerical," to use the term that has been popularized by Knuth [8]) rather than analytic.

Fast algorithms for polynomial and power series division were later in coming (by some six years). The first speedup of polynomial division was an $O(n \log^2 n)$ algorithm due to Borodin and Moenck [1]. Independently, Seiveking [13] devised an $O(n \log n)$ algorithm for power series inversion. Strassen [14, Sec. 3.5], by reducing polynomial division to power series inversion (the analogue of the reduction of integer quotient-remainder division to integer inversion mentioned above), was able to apply Seiveking's power series inversion algorithm to produce an improved [$O(n \log n)$] polynomial division algorithm. These algorithms all appealed to the $O(n \log n)$ fast Fourier transform based algorithm for multiplying polynomials of degree n (e.g., see [2, Ch. 4]).

An important conceptual simplification was made by Kung [9] who showed that Seiveking's algorithm for inverting a power series a(t) was essentially Newton's method applied over a power series domain to the equation $f(x) = 1/x - a(t) = 0$. A further measure of coherence and unity was provided by Moenck and Allen [10], who studied the problem of division in the general setting of an abstract Euclidean domain. They were able to devise an abstract division algorithm that yielded as special cases both the integer division algorithm of Cook and the polynomial division algorithm of Seiveking and Strassen.

This present work is somewhat orthogonal to the above developments (though influenced by all of them, especially the results of Kung). Rather than deal exclusively with a particular problem, such as division, we shall consider the problem of root-finding in general over a power series domain. By showing the general applicability of Newton's method for handling this problem, our goal is to set the stage for a variety of applications.

2. MATHEMATICAL PRELIMINARIES

In Section 2.1 we formulate in algebraic terms the notions of approximation and convergence in F[[t]]. In Section 2.2 we use the well-known idea of a formal derivative to obtain a Taylor-like series expansion for bivariate polynomials in D[x][y]. All this is preparatory to our goal of analyzing Newton's method in the algebraic setting of solving polynomial equations over a power series domain.

2.1. Approximation and convergence in F[[t]]

Let $a(t) = \sum_{i=0}^{\infty} a_i t^i$, $b(t) = \sum_{i=0}^{\infty} b_i t^i$, etc., be power series in F[[t]].

Congruence mod t^m. The relation of congruence ($\equiv$) mod t^m is defined in F[[t]] by

$$(2.1) \quad a(t) \equiv b(t) \pmod{t^m} \iff a_i = b_i \quad \text{for } i = 0, 1, \ldots, m-1,$$

i.e., two formal power series are congruent mod t^m if their first m terms are the same.

The reduction mod t^m operator is defined on F[[t]] by

$$(2.2) \quad a(t) \bmod t^m = \sum_{i=0}^{m-1} a_i t^i.$$

$a(t) \bmod t^m$ is seen to be the unique polynomial of degree $< m$ that is congruent mod t^m to a(t).

By direct proof or by appealing to the fact that congruence mod t^m is indeed a congruence relation, we obtain

Proposition 2.1. (Computation mod t^m.)

$$[a(t) \stackrel{+}{\times} b(t)] \bmod t^m = [a(t) \bmod t^m \stackrel{+}{\times} b(t) \bmod t^m] \bmod t^m.$$

From Proposition 2.1 we conclude the following simple procedure for computing mod t^m in F[[t]]: carry out power series addition, subtraction, and multiplication as the case may be, but in all operands and results of operations retain only those terms having degree $< m$. (Note 2.)

Approximation and convergence. The order of $a(t) \in F[[t]]$, ord a(t), is defined by

$$(2.3) \quad \text{ord } a(t) = k \text{ if } a_k \neq 0 \text{ and } a_i = 0 \quad \text{for } i < k.$$

The zero power series is assigned order $+\infty$.

We shall need the following easily proven facts about order, which we record as

Proposition 2.2.

$$\text{ord}[a(t) + b(t)] \geq \min[\text{ord } a(t), \text{ord } b(t)];$$

$$\text{ord}[a(t)b(t)] = \text{ord } a(t) + \text{ord } b(t).$$

If $a(t) \equiv b(t) \pmod{t^m}$ then by (2.1) $b(t)$ agrees with $a(t)$ in its first m terms, and we accordingly regard $b(t)$ as an $O(t^m)$ ("order t^m") approximation to $a(t)$. $a(t) \bmod t^m$ then takes the form of a distinguished $O(t^m)$ approximation to $a(t)$ -- the unique $O(t^m)$ polynomial, or truncated power series, approximation to $a(t)$ having degree $< m$. (Note 2.)

We note that $O(t^m)$ approximation and order are trivially related by

Proposition 2.3.

$b(t)$ is an $O(t^m)$ approximation to $a(t)$ $\iff \text{ord}[a(t) - b(t)] \geq m$.

Remark. $e(t) = a(t) - b(t)$ can be regarded as the error in approximating $a(t)$ by $b(t)$. (But note that this "error" is a formal power series.)

With the idea of approximation in hand, it makes sense to talk about convergence of a sequence of (formal) power series. Let $a(t)$ be a power series and let $\{a^{(k)}(t)\}_{k=0}^{\infty}$ be a sequence of power series in $F[[t]]$. Define $e^{(k)}(t) = a(t) - a^{(k)}(t)$. We say that the sequence $\{a^{(k)}(t)\}_{k=0}^{\infty}$ converges to $a(t)$ if for any integer $m > 0$ there exists an integer $K \geq 0$ such that $\text{ord } e^{(k)} \geq m$ for all $k \geq K$; i.e., for all sufficiently large k, $a^{(k)}(t)$ is an $O(t^m)$ approximation to $a(t)$.

We shall not only be interested in convergence, but in fast convergence. With this in mind we say that the sequence $\{a^{(k)}(t)\}_{k=0}^{\infty}$ of formal power series converges geometrically to $a(t)$ if there exist constants $c > 0$ and $p > 1$ such that $\text{ord } e^{(k)} \geq cp^k$ for all $k \geq 0$. The greatest such constant p is called the order of the (geometric) convergence (yet another use of the term "order").

2.2. Formal derivatives and Taylor series expansions

The (formal) derivative of a polynomial $f(x) = \sum_{i=0}^{\infty} f_i x^i$ in $D[x]$ is defined by

$$f'(x) = \sum_{i=0}^{\infty} (i+1) f_{i+1} x^i. \tag{2.4}$$

It can be shown by completely algebraic arguments that formal derivatives satisfy the familiar properties of analytic derivatives:

$$\begin{aligned} &f(x) = g(x) + h(x) \\ &\Rightarrow f'(x) = g'(x) + h'(x), \\ &f(x) = g(x)h(x) \\ &\Rightarrow f'(x) = g(x)h'(x) + g'(x)h(x), \\ &f(x) = g(h(x)) \\ &\Rightarrow f'(x) = g'(h(x))h'(x). \end{aligned} \tag{2.5}$$

We now wish to apply this concept of a formal derivative in the setting of a bivariate polynomial domain $D[x][y]$.

We identify $D[x][y]$ -- the set of all polynomials $\sum_{j=0}^{\infty} a_j(x) y^j$ with coefficients $a_j(x) = \sum_{i=0}^{\infty} a_{ij} x^i \in D[x]$ -- with $D[x, y]$ -- the set of all polynomials $\sum_{i,j=0}^{\infty} a_{ij} x^i y^j$ in two "simultaneous" indeterminates x and y (only finitely many of the a_{ij}'s are nonzero). The identification is the obvious one: $\sum_{j=0}^{\infty} a_j(x) y^j$ is identified with $\sum_{i,j=0}^{\infty} a_{ij} x^i y^j$, and we denote either by $a(x, y)$.

Remark. This identification of polynomial forms (expressions) is compelling when we interpret such bivariate polynomial forms as polynomial functions $E \times E \to E$, where E is any extension of D. Since the form $\sum_{j=0}^{\infty} a_j(x) y^j$ can be converted to the form $\sum_{i,j=0}^{\infty} a_{ij} x^i y^j$ by finitely many applications of the postulates for a commutative ring, these forms are equal as polynomial functions.

The next theorem is crucial for our purposes.

Theorem 2.4. (Formal Taylor series.) Let $f(x)$ be a polynomial in $D[x]$. Then in $D[x][y]$

$$f(x+y) = f(x) + f'(x)y + g(x, y)y^2$$

for some $g(x, y) \in D[x, y]$.

Proof. We note first that $x+y$ is a polynomial in $D[x][y]$. Therefore the value of $f(x)$ evaluated at $x+y$ is a polynomial that is also in $D[x][y]$, say

$$(*) \quad f(x+y) = f_0(x) + f_1(x)y + g(x, y)y^2$$

for some $f_0(x), f_1(x) \in F[x]$ and $g(x, y) \in D[x][y]$.

Letting $y = 0$ in (*) yields $f(x) = f_0(x)$. Taking formal derivatives (in $D[x][y]$) of both sides of (*) (appealing to properties (2.5)) gives

$$(**) \quad f'(x+y) = f_1(x) + 2yg(x, y) + g'(x, y)y^2$$

($g'(x, y)$ = the derivative of $g(x, y)$ in $D[x][y]$). Letting $y = 0$ in (**) yields $f'(x) = f_1(x)$, completing the proof of the theorem. (Note 3.)

Example. Consider $f(x) = x^4 \in \underline{Z}[x]$. Then in $\underline{Z}[x][y]$ we have

$$\begin{aligned} f(x+y) &= (x+y)^4 \\ &= x^4 + 4x^3y + 6x^2y^2 + 4xy^3 + y^4 \\ &= f(x) + f'(x)y + g(x, y)y^2 \end{aligned}$$

(for $g(x, y) = 6x^2 + 4xy + y^2$).

In the next (the main) section we combine the two sets of ideas of Sections 2.1 and 2.2.

3. NEWTON'S METHOD OVER A POWER SERIES DOMAIN

We now establish the validity and efficacy of Newton's method as a root-finding algorithm over a power series domain.

Theorem 3.1. (Main result.) Let $f(x) \in D[x]$ be a polynomial with coefficients in a power series domain $D = F[[t]]$. Let $\alpha \in F$ be an $O(t)$ approximation to a root $\overline{x} = \sum_{i=0}^{\infty} \overline{x}_i t^i$ of $f(x)$ (i.e., $\alpha = \overline{x}_0$), so that

$$f(\alpha) \bmod t = 0 \qquad \text{(Condition 1)}.$$

Further suppose that α satisfies

$$f'(\alpha) \bmod t \neq 0 \qquad \text{(Condition 2)}.$$

Then the sequence of iterates $x^{(0)} = \alpha$, $x^{(1)}, x^{(2)}, x^{(3)}, \ldots$ computed in $F[[t]]$ according to Newton's iteration

$$(3.1) \quad x^{(k+1)} = [x^{(k)} - f(x^{(k)})f'(x^{(k)})^{-1}] \bmod t^{2^{k+1}}$$

is such that $x^{(k)} = x^{(k)}(t) = \sum_{i=0}^{2^k-1} x_i^{(k)} t^i$ is an $O(t^{2^k})$ approximation to $\overline{x}$.

Remarks: 1. Theorem 3.1 asserts that Conditions 1 and 2 assure quadratic convergence of Newton's method ($p = 2$ and $c = 1$ in our definition of geometric convergence given at the end of Section 2.1). And in our power series setting quadratic convergence implies that the number of correct terms exactly doubles with each successive iterate.

2. Just as in the numerical case, the power series application of Newton's method requires that the initial approximation, $x^{(0)}$, to the desired root $\overline{x}$ be "close" to $\overline{x}$ for convergence to be assured. The above theorem gives a sharp global convergence criterion that involves a polynomial only over the ground field, F, rather than over the power series domain $F[[t]]$: If α is a simple root of the polynomial $f(x) \bmod t \in F[x]$ (i.e., if α satisfies Conditions 1 and 2 above), then quadratic convergence to a root $\overline{x} \in F[[t]]$ with constant term $\overline{x}_0 = \alpha$ is assured.

We interpolate an example to illustrate the force of these conditions. Consider $f(x) = x^2 + tx - 4$ considered as a polynomial over $\underline{Q}[[t]]$. Now $\alpha = 2$ is a root of $f(x) \bmod t = x^2 - 4$; moreover, $\alpha = 2$ is simple root of $f(x) \bmod t$, since $f'(x) \bmod t = 2x \neq 0$ at $x = 2$. Theorem 3.1 then promises that Newton's method (3.1) converges quadratically to the root $\overline{x}$ of $f(x)$ with $x^{(0)} = 2$ (which in this case happens to be $\overline{x} = 2 - t/2 + t^2/16 - t^4/1024 + t^6/32278 + - \ldots$).

3. The computation of (3.1) is carried out mod $t^{2^{k+1}}$, as indicated. According to Proposition 2.1 and its ensuing discussion, this means that all of the indicated power series operations can be carried out mod $t^{2^{k+1}}$ -- only terms of degree $< 2^{k+1}$ need be retained. For example, $f'(x^{(k)})^{-1} \bmod t^{2^k}$ is obtained by first computing $f'(x^{(k)}) \bmod t^{2^k}$ as a

truncated power series $g(t) = \sum_{i=0}^{2^k-1} g_i t^i$ and then returning the first 2^k-1 terms of $g(t)^{-1}$ (Note 1).

Proof of Theorem 3.1. We are to show that $x^{(k)}$ is an $O(t^{2^k})$ approximation to $\bar{x}$ ($k = 0, 1, 2, \ldots$). The proof is by induction on k.

The basis holds trivially: By Condition 1, $x^{(0)} = \alpha$ is an $O(t)$ approximation to $\bar{x}$.

As for the induction step, assume for $k \geq 0$ that $x^{(k)}$ is an $O(t^{2^k})$ approximation to $\bar{x}$. Letting $\bar{x} = x^{(k)} + e^{(k)}$, this means (Proposition 2.3) that ord $e^{(k)} \geq 2^k$ ($e^{(k)}$ = the error in the k-th iterate, $x^{(k)}$).

Now, expand the given polynomial $f(x) \in D[x]$ ($D = F[[t]]$) as a Taylor series in $D[x][y]$ according to Theorem 2.4:

$$f(x+y) = f(x) + f'(x)y + g(x, y)y^2.$$

Regarding $f(x+y)$ above as a polynomial function of two variables x and y over $F[[t]]$, we evaluate $f(x+y)$ by substituting $x^{(k)} \in F[[t]]$ for x and $e^{(k)} \in F[[t]]$ for y. This gives

$$\begin{aligned} 0 &= f(\bar{x}) \\ &= f(x^{(k)} + e^{(k)}) \\ (*) \qquad &= f(x^{(k)}) + f'(x^{(k)})e^{(k)} + e^{(k)^2}v, \end{aligned}$$

where $v = g(x^{(k)}, e^{(k)}) \in F[[t]]$.

We claim now that $f'(x^{(k)})$ is invertible in $F[[t]]$. Proof of claim: Since ord $e^{(k)} > 0$, it follows that $x_0^{(k)} = \bar{x}_0 = \alpha$. Hence $f'(x_0^{(k)}) \bmod t = f'(\alpha) \bmod t \neq 0$, the last inequality being Condition 2. But $f'(x_0^{(k)}) \bmod t$ is nothing more than the constant term of $f'(x^{(k)}) \in F[[t]]$. Its non-vanishing is precisely the condition that $f'(x^{(k)})$ be invertible in $F[[t]]$ (Note 1). This completes the proof of the claim.

Multiplying through by $f'(x^{(k)})^{-1} \in F[[t]]$ in (*) and transposing yield

$$e^{(k)} = -f(x^{(k)})f'(x^{(k)})^{-1} + e^{(k)^2}vf'(x^{(k)})^{-1},$$

or

$$\begin{aligned} \bar{x} &= [x^{(k)} - f(x^{(k)})f'(x^{(k)})^{-1}] \\ &\quad + e^{(k)^2}vf'(x^{(k)})^{-1} \\ &= [x^{(k)} - f(x^{(k)})f'(x^{(k)})^{-1}] \bmod t^{2^{k+1}} \\ &\quad + u + e^{(k)^2}vf'(x^{(k)})^{-1} \\ &\qquad\qquad\qquad\qquad (\text{ord } u \geq 2^{k+1}) \\ &= x^{(k+1)} + \{u + e^{(k)^2}vf'(x^{(k)})^{-1}\}. \end{aligned}$$

Now, ord $u \geq 2^{k+1}$ as noted, and ord $e^{(k)} \geq 2^k$ by our induction hypothesis. Hence by Proposition 2.2, ord $\{\ldots\} \geq 2^{k+1}$. Thus ord $\bar{x} - x^{(k+1)} \geq 2^{k+1}$, which makes $x^{(k+1)}$ an $O(t^{2^{k+1}})$ approximation to the root $\bar{x}$. This completes the proof by induction of Theorem 3.1.

4. COMPUTATIONAL ASPECTS

We first record some preliminary results related to bounds on computing times that we shall need for our analysis of Newton's method as a polynomial root-finder over a power series domain.

Let T(N) be a bound on the time required by Newton's method to compute the first N terms of a root $\bar{x}(t) = \sum_{i=0}^{\infty} \bar{x}_i t^i$ of $f(x) = 0$.

Lemma 4.1. T(N) satisfies

$$\begin{aligned} (4.1) \qquad T(2N) &= T(N) + F(N) \\ T(1) &= 0 \end{aligned}$$

where F(N) is the time required to compute the first 2N terms of $\bar{x}(t)$ from the previously computed first N terms.

The recursion of Lemma 4.1 is an immediate consequence of Newton's iteration (3.1), which expresses the computation of $x^{(k+1)}$ -- the first 2^{k+1} terms of $\bar{x}$ -- as a function of $x^{(k)}$ -- the first 2^k terms of $\bar{x}$. The basis, $T(1) = 0$, is a statement of our tacit assumption that the acquisition of the initial approximation $x^{(0)} = \bar{x}_0$ lies outside the scope of Newton's method proper.

Lemma 4.2. Let γ be a positive real number.

(a) If $F(N) = O(N^\gamma)$ then $T(N) = O(N^\gamma)$;
(b) If $F(N) = O(N^\gamma \log N)$ then $T(N) = O(N^\gamma \log N)$.

Sketch of proof. Let $N = 2^n$. From Eq. (4.1) we have

$$T(2^n) = T(2^{n-1}) + F(2^{n-1}).$$

Iterating this relationship n - 1 times and using $T(1) = 0$ give

$$T(2^n) = \sum_{k=0}^{n-1} F(2^k).$$

Substituting $F(2^k) = c(2^\gamma)^k$ (c a real constant) readily yields part (a) of Lemma 4.2; substituting $F(2^k) = ck(2^\gamma)^k$ leads to part (b). We omit the straightforward details.

Remark. Lemma 4.2 conveys an interesting algorithmic consequence of quadratic convergence, at least for our particular choices of F(N): The order of magnitude time bound for Newton's method is determined solely by the time required for the last iteration.

We shall also need bounds on the times required to compute the first N terms of the product of two truncated power series $a(t) = \sum_{i=0}^{N-1} a_i t^i$, $b(t) = \sum_{i=0}^{N-1} b_i t^i$ in F[[t]] (i) using the "classical" (= convolution) algorithm -- call this $M_C(N)$ -- and (ii) using the fast Fourier transform (FFT) algorithm -- call this bound $M_F(N)$. We record these well-known bounds as

Lemma 4.3. $M_C(N) = O(N^2)$;

$$M_F(N) = O(N \log N).$$

From Note 1 we see that the bound $M_C(N)$ also applies to computing the first N terms of the inverse $1/a(t)$ ($a_0 \neq 0$).

4.1. r-th root extraction; inversion

To compute an r-th root of $a = a(t) = \sum_{i=0}^{\infty} a_i t^i \in F[[t]]$, we take $f(x) = x^r - a$. If the constant term a_0 has an r-th root $\alpha \neq 0$ in F, then Conditions 1 and 2 of Theorem 3.1 are evidently satisfied. It follows that Newton's iteration (3.1), which now becomes

$$(4.2) \quad x^{(k+1)} = [(r-1)x^{(k)} + a/x^{(k)^{r-1}}]/r \mod t^{2^{k+1}},$$

converges quadratically to an r-th root $\bar{x}(t) = \sum_{i=0}^{\infty} \bar{x}_i t^i$ with $\bar{x}_0 = \alpha$.

If classical power series multiplication and inversion algorithms are used for computing $x^{(k+1)}$ according to (4.2), then F(N) of Eq. (4.1) becomes $O((\log r)N^2))$ by Lemma 4.3, and by Lemma 4.2 we have

Proposition 4.4. The time for Newton's method to compute the first N terms of an r-th root of $a(t) \in F[[t]]$ using classical power series multiplication and division is bounded by $O((\log r)N^2)$.

When $r = -1$ we have $f(x) = 1/x - a$, so that a root $\bar{x}$ of f(x) is an inverse of a(t). Unfortunately our main Theorem 3.1 does not apply to this case because $f(x) = 1/x - a$ is not a polynomial. This behooves us to give a direct proof of quadratic convergence of (4.2) for the case $r = -1$.

Lemma 4.5. Let $a = a(t) \in F[[t]]$ have constant term $a_0 \neq 0$. Then with $x^{(0)} = 1/a_0$ the iteration

$$(4.3) \quad x^{(k+1)} = [2x^{(k)} - ax^{(k)^2}] \mod t^{2^{k+1}}$$

converges quadratically to $\bar{x} = 1/a$.

Proof. Let $e^{(k)} = \bar{x} - x^{(k)}$, where $\bar{x} = 1/a$. By (4.3) we have

$$x^{(k+1)} = 2(\bar{x} - e^{(k)}) - a(\bar{x} - e^{(k)})^2$$

$$= \bar{x} - ae^{(k)^2},$$

giving

$$e^{(k+1)} = ae^{(k)^2}.$$

Quadratic convergence of (4.3) now follows exactly as in Theorem 3.1.

By Lemmas 4.1-3, we conclude

Theorem 4.6. (Sieveking [13], Kung [9].) The time for Newton's method (specifically, iteration (4.3)) to compute the first N terms of $1/a \in F[[t]]$ using fast (FFT) power series multiplication is bounded by O(N log N).

Returning to our problem of computing an r-th root, with fast division in hand we can improve Proposition 4.4 to

Theorem 4.7. If fast [O(N log N)] algorithms are used for power series multiplication and inversion, then the time for Newton's method to compute the first N terms of an r-th root of $a(t) \in F[[t]]$ is bounded by $O((\log r)N \log N)$.

4.2. Analysis of the general case

Let Newton's iteration (3.1)

$$x^{(k+1)} = [x^{(k)} - f(x^{(k)})f'(x^{(k)})^{-1}] \mod t^{2^{k+1}}$$

be applied for the general case of a polynomial $f(x) = \sum_{i=0}^{r} f_i x^i$ of degree r, with coefficients $f_i \in F[[t]]$. To obtain $x^{(k+1)}$ we first require the evaluation of $v = f(x^{(k)}) \mod t^{2^{k+1}}$ and $w = f'(x^{(k)}) \mod t^{2^{k+1}}$, which can be computed according to Horner's method for the simultaneous evaluation of a polynomial and its first derivative:

(4.4) (Horner's method over F[[t]])

Initialization: $v \leftarrow f_r \mod t^{2^k}$;

$w \leftarrow 0$;

Iteration: For $i \leftarrow r-1, r-2, \ldots, 0$:

$w \leftarrow [v + x^{(k)} \times w] \mod t^{2^k}$;

$v \leftarrow [f_i + x^{(k)} \times v] \mod t^{2^k}$.

With v and w in hand, $x^{(k+1)}$ is then computed according to

(4.5) $x^{(k+1)} \leftarrow [x^{(k)} - w \times v^{-1}] \mod t^{2^k}$.

A bound on the time to compute $x^{(k+1)}$ according to (4.4) and (4.5) is readily determined. 2r multiplicative power series operations are performed. If classical multiplication and division algorithms are used, then F(N) of Lemma 4.1 becomes $O(rN^2)$. If fast multiplication and division algorithms are used (where we appeal to Theorem 4.6 for fast division), then F(N) of Lemma 4.1 becomes O(rN log N). From Lemma 4.2 we conclude

Theorem 4.8. (Computing time bound for Newton's method.) The time for Newton's method (specifically (4.4) and (4.5)) to compute the first N terms of a root $\bar{x} = \sum_{i=0}^{\infty} \bar{x}_i t^i$ of $f(x) = \sum_{i=0}^{r} f_i x^i$ ($f_i \in F[[t]]$) for specified $\bar{x}_0$ is bounded by

(a) $O(rN^2)$ if classical power series multiplication and division are used;

(b) O(rN log N) if fast power series multiplication and division are used.

4.3. Sample applications

We now present three sample applications of a straightforward (quick and dirty) implementation of Newton's method (3.1) using FORMAC [15]. This implementation operates over the power series domain Q[[t]], thus exploiting FORMAC's rational mode arithmetic capability.

Problem 1. (Square root.) Compute a square root of $a = a(t) = 1 + t + 2t^2 + 3t^3 + \ldots$.

Solution. Applying three iterations of Newton's method to $f(x) = x^2 - a$ with initial approximation $x^{(0)} = 1$ yields an $O(t^8)$ approximation to the desired root:

```
X(0) = 1
< E(K) = X(K) - X(K-1) >
E(1) = 1/2 T
E(2) = 7/8 T**2 + 17/16 T**3
E(3) = 139/128 T**4 + 263/256 T**5 + 995/
1024 T**6 + 1969/2048 T**7
```

Problem 2. (Solution of a cubic.) Compute a root of

$$f(x) = x^3 - \frac{2}{1-t}x + \beta$$

for (i) $\beta = 1$ and (ii) $\beta = 7/8$.

Solution. (i) $\beta = 1$: $x^{(0)} = 1$ satisfies Conditions 1 and 2 of Theorem 3.1. Three iterations of Newton's method yield:

```
X(0) = 1
< E(K) = X(K) - X(K-1) >
E(1) = 2 T
E(2) = - 6 T**2 + 58 T**3
E(3) = - 622 T**4 + 7506 T**5 - 96822 T**6
 + 1307466 T**7
```

(ii) $\beta = 7/8$: $x^{(0)} = 1/2$ satisfies Conditions 1 and 2 of Theorem 3.1. Three iterations of Newton's method yield:

X(0) = 1/2

< E(K) = X(K) - X(K-1) >

E(1) = - 4/5 T

E(2) = 156/125 T^2 - 13508/3125 T^3

E(3) = 273196/15625 T^4 - 153756036/1953125 T^5 + 18478847772/48828125 T^6 - 2322598397508/1220703125 T^7

Our final application involves the problem of power series reversion. The reversion of a power series $a(t) = \sum_{i=1}^{\infty} a_i t^i \in F[[t]]$ (constant term = 0) is the power series $x(t) = \sum_{i=1}^{\infty} x_i t^i$ that satisfies $a(x(t)) = t$. Thus $x(t)$ is the inverse of $a(t)$ with respect to the operation of composition of power series. (The existence and uniqueness of $x(t)$ follows readily from group theoretic considerations; e.g., see Henrici [7, Sec. 1.7].)

We observe that the reversion of $a(t)$ is a root of

$$f(x) = a(x) - t, \tag{4.6}$$

so we naturally consider Newton's method for its solution. But $f(x)$ is a power series, rather than a polynomial, in x, so it would seem that the solution of (4.6) lies outside our polynomial root-finding framework. This setback, however, is illusory, since only the first n terms of $a(t)$ enter into the computation of the first n terms of its reversion. [Proof: Since ord $x(t) \geq 1$, it follows by Proposition 2.2 that ord $x(t)^k > N$ for all $k > N$, whence $a(x(t)) \equiv a^{(N)}(x(t)) \pmod{t^{N+1}}$ where $a^{(N)}(t) = a(t) \bmod t^{N+1} = \sum_{i=1}^{N} a_i t^i$.] In (4.6) we may therefore replace $a(x)$ by $a(x) \bmod t^{N+1}$, where N is the number of terms of the reversion we wish to compute. This renders $f(x)$ as a bona fide polynomial, and Newton's method accordingly applies. (Note 4.)

Problem 3. Compute an $O(t^{16})$ approximation to tan t by reverting the power series for

$$\arctan t = t - t^3/3 + t^5/5 - + \ldots$$

Solution. Four iterations of Newton's method applied to (4.6) with $a(t) = \arctan t$ and $x^{(0)} = 0$ yield the desired solution.

X(0) = 0

< E(K) = X(K) - X(K-1) >

E(1) = T

E(2) = 1/3 T^3

E(3) = 2/15 T^5 + 17/315 T^7

E(4) = 62/2835 T^9 + 1382/155925 T^11 + 21844/6081075 T^13 + 929569/638512875 T^15

---

5. SUMMARY

We have seen (Section 2) that the analytic concepts of approximation, convergence, differentiation, and Taylor series expansion can all be interpreted in the algebraic setting of a power series domain F[[t]] over an abstract field F. This permitted us to establish in algebraic terms the applicability of Newton's method to computing a (power series) root of a polynomial with power series coefficients. In particular, Newton's method operating over a power series domain has been shown (Theorem 3.1) to be quadratically convergent subject to the initial approximation being "close" to the desired root -- quite analogous to its behaviour as a numerical algorithm. Our "closeness" conditions 1 and 2 of Theorem 3.1 were sharp, involving the solution of a polynomial equation over the ground field F (rather than over F[[t]]).

In Section 4 we investigated the computational aspects of Newton's method. Theorem 4.8 shows that Newton's method, in conjunction with fast power series multiplication and division, yields a fast root-finding algorithm [$O(rN \log N)$ for finding the first N terms of a root of an r-th degree polynomial vs. $O(rN^2)$ for "classical" approaches].

The usefulness of a power series manipulation capability in symbolic mathematics is being increasingly recognized (e.g., see

[11]). In a symbolic mathematics system setting, we feel that Newton's method offers a unified and easily implemented algorithm for solving a wide variety of power series manipulation problems that can be cast into a root-finding mold. Our experience with a FORMAC implementation (Section 4.3) attests to the practicality of Newton's method being used in this way. (The efficiency of Newton's method is probably of secondary practical importance, since a serious user is more likely to want three terms of a power series than three thousand.)

In conclusion: Newton's method has long since gained the status of a great numerical algorithm. We feel that it is a great algebraic algorithm as well.

NOTES

1. The classical power series division algorithm for computing $c(x) = a(x)/b(x)$ is obtained by equating coefficients in $a(x) = b(x)c(x)$:

$$c_0 = a_0 b_0^{-1}$$

$$c_k = b_0^{-1}(a_k - \sum_{i=1}^{k} b_i c_{k-i}) \qquad (k = 1, 2, \ldots).$$

(The above formula clearly indicates the necessary and sufficient condition that power series division be possible: the constant term of the divisor power series must be invertible in the coefficient domain.)

2. Congruence mod t^m over the power series domain $F[[t]]$ is the algebraic analogue of the familiar congruence mod m relation over the domain $\underline{Z}$ of integers: For $a, b \in \underline{Z}$ and $a(t), b(t) \in F[[t]]$, and for m a nonnegative integer, we have

$$a \equiv b \pmod{m} \quad \Leftrightarrow \quad a - b = km$$

$$a(t) \equiv b(t) \pmod{t^m} \quad \Leftrightarrow \quad a(t) - b(t) = k(t)t^m$$

for some $k \in \underline{Z}$, $k(t) \in F[[t]]$.

The set $F_m[[t]]$ of all truncated power series (= polynomials) having degree < m is easily shown to be a ring under mod t^m addition and multiplication. This ring is the analogue of the ring $\underline{Z}_m = \{0, 1, \ldots, m-1\}$ of integers mod m under mod m addition and multiplication.

3. A more general Taylor's formula than the one of Theorem 2.4 can be established (e.g., see [5, Sec. 30]). However, our development, besides being sufficient for our purposes, has the advantage of not imposing any restrictions on the characteristic of the field F.

4. From Theorem 4.8 we have the computation of an $O(t^N)$ approximation to the reversion of $a(t) = \sum_{i=1}^{\infty} a_i t^i$ by Newton's method requires (a) time $O(N^3)$ if classical power series multiplication and division are used and (b) time $O(N^2 \log N)$ if fast power series multiplication and division are used. Thus if classical power series operations are used, the same $O(N^3)$ time bound is achieved by Newton's method as is achieved by "standard" reversion algorithms such as discussed in Knuth [8, Sec. 4.7].

Recently Brent and Kung [3] have devised an even faster than $O(N^2 \log N)$ reversion algorithm by combining Newton's method with an ingenious fast composition algorithm.

(This author owes the original idea of using Newton's method to compute power series reversions to Dr. Richard Jenks, circa February 1974.)

REFERENCES

[1] Borodin, A. and Moenck, R. (1974) Fast modular transforms. J. Comput. System Sci. 8: 366-386.

[2] Borodin, A. and Munro, I. (1975) The computational complexity of algebraic and numeric problems. American Elsevier, New York.

[3] Brent, R.P. and Kung, H.T. (1976) $O((n \log n)^{3/2})$ algorithms for composition and reversion of power series. In: Analytic computational complexity, J.F. Traub (ed.). Academic Press, New York.

[4] Cook, S.A. (1966) On the minimum computation time of functions. Ph.D. thesis, Harvard U.

[5] Godement, R. (1968) Algebra. Houghton Mifflin, Boston.

[6] Henrici, P. (1964) Elements of numerical analysis. Wiley, New York.

[7] Henrici, P. (1974)
Applied and computational complex analysis (Vol. 1).
Wiley, New York.

[8] Knuth, D.E. (1969)
The art of computer programming, Vol. 2 (Seminumerical algorithms).
Addison-Wesley, Reading, Mass.

[9] Kung, H.T. (1974)
On computing reciprocals of power series.
Numer. Math. 22: 341-348.

[10] Moenck, R. and Allen, J. (1975)
Efficient division algorithms in Euclidean domains.
Research report CS-75-18, U. of Waterloo.

[11] Norman, A.C. (1975)
Computing with formal power series.
ACM Trans. Math. Software 1: 346-356.

[12] Schönhage, A. and Strassen, V. (1971)
Schelle Multiplikation grosser Zahlen (Fast multiplication of large integers).
Computing 7: 281-292.

[13] Sieveking, M. (1972)
An algorithm for division of power series.
Computing 10: 153-156.

[14] Strassen, V. (1973)
Die Berechnungskomplexität von elementarsymmetrischen Funktionen und von Interpolationskoeffizienten.
Numer. Math. 20: 238-251.

[15] Tobey, R., et al (1969)
PL/I FORMAC symbolic mathematics interpreter.
IBM Contributed Program Library, 360D-03.3.004, Hawthorne, N.Y.

ADDENDUM

The interested reader is referred to the following highly relevant work, which came to my attention after completion of the manuscript: the work of H.T. Kung and J.F. Traub on fast algorithms for algebraic functions (see Program for Symposium on New Directions and Recent Results in Algorithms and Complexity, Apr. 7-9, 1976, Carnegie-Mellon University, p. 75); and the work of D.Y.Y. Yun on algebraic algorithms using p-adic constructions (these proceedings; see also Yun's paper in Analytic computational complexity [3]).

On the Subresultant PRS Algorithm

W. S. Brown

Bell Laboratories,
Murray Hill, New Jersey 07974

ABSTRACT

This paper is a sequel to two earlier papers [1, 2] on the generalization of Euclid's algorithm to domains of polynomials.

In attempting such a generalization one easily arrives at the concept of a *polynomial remainder sequence* (PRS), and then quickly discovers the phenomenon of explosive coefficient growth.

Fortunately, this explosive growth is not inherent in the problem, but is only an artifact of various naive solutions. If one removes the *content* (that is, the greatest common divisor of the coefficients) from each polynomial in a PRS, the coefficient growth in the resulting *primitive PRS* is relatively modest. However, the cost of computing the content (by applying Euclid's algorithm in the coefficient domain) may be unacceptably or even prohibitively high, especially if the coefficients are themselves polynomials in one or more additional variables.

The key to controlling coefficient growth without the costly computation of greatest common divisors (GCD's) of coefficients is the discovery by Collins [3] that every polynomial in a PRS is proportional to some subresultant of the first two. By arranging for the constants of proportionality to be unity, Collins developed the *subresultant PRS algorithm,* which is the subject of this paper. Unfortunately, Collins' formulation of the algorithm was too complicated for convenient application, and he therefore recommended the simpler *reduced PRS algorithm* as a practical compromise.

Later, Brown and Traub [1] discovered the *fundamental theorem of subresultants,* and used it to derive a much simpler formulation of the subresultant PRS algorithm. Also, Brown [2] derived essentially linear bounds on the coefficient growth in a subresultant PRS, while showing that the coefficient growth in a reduced PRS can be exponential if the sequence involves degree differences greater than unity. Although such *abnormal* sequences are a set of measure zero in the space of all PRS's, they are not uncommon in practice, and it is important to deal sensibly with them when they arise.

A few months after [1] and [2] were published, I discovered a corollary of the fundamental theorem, which led to a simpler derivation and deeper understanding of the subresultant PRS algorithm. The new approach, which is presented in this paper, reveals the subresultant PRS algorithm as a simple generalization of the reduced PRS algorithm, and converts the conjecture that was mentioned in [1] and [2] into an elementary remark.

Although I cannot assert with confidence that the subresultant PRS algorithm is optimal for any important class of problems, it is clearly the best of its kind and deserves to be thoroughly understood. Among its competitors are the *modular GCD algorithm,* which is shown in [2] to be superior if the given polynomials are sufficiently large and sufficiently dense, and the *EZ-GCD algorithm* of Moses and Yun [4], which is also modular, but has the advantage of benefiting from sparseness. On the other hand, if one desires only the resultant of the given polynomials, and their degrees are not too large, it may be advantageous to evaluate Sylvester's determinant, or the equivalent but lower-order Bezout's determinant, via expansion by minors. The merits of this approach are explored empirically by Ku and Adler [5], and their important but overstated conclusions are challenged by Collins [6].

In this paper the subresultant PRS algorithm is presented from the new viewpoint, and the outstanding conjecture is proved. The algorithm is then analyzed, and its practical importance is assessed.

---

[1] W. S. Brown and J. F. Traub, "On Euclid's Algorithm and the Theory of Subresultants", *J. ACM* **18,** pp. 505-514 (October 1971).

[2] W. S. Brown, "On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors", *J. ACM* **18,** pp. 478-504 (October 1971).

[3] G. E. Collins, "Subresultants and Reduced Polynomial Remainder Sequences", *J. ACM* **14,** pp. 128-142 (1967).

[4] Joel Moses and David Y. Y. Yun, "The EZ GCD Algorithm", *Proc. of the ACM National Conference,* pp. 159-166 (August 1973).

[5] S. Y. Yu and R. J. Adler, "Computing Polynomial Resultants: Bezout's Determinant vs. Collins' Reduced PRS Algorithm", *Comm. ACM* **12,** pp. 23-30 (January 1969).

[6] G. E. Collins, "Comment on a Paper by Ku and Adler", Letter to the Editor, *Comm. ACM* **12,** pp. 302-303 (June 1969).

Polynomial Real Root Isolation Using Descarte's Rule of Signs*

George E. Collins
and
Alkiviadis G. Akritas
Computer Sciences Department, University of Wisconsin
1210 W. Dayton Street, Madison, Wisconsin 53706

Abstract

Uspensky's 1948 book on the theory of equations presents an algorithm, based on Descartes' rule of signs, for isolating the real roots of a squarefree polynomial with real coefficients. Programmed in SAC-1 and applied to several classes of polynomials with integer coefficients, Uspensky's method proves to be a strong competitor of the recently discovered algorithm of Collins and Loos. It is shown, however, that it's maximum computing time is exponential in the coefficient length. This motivates a modification of the Uspensky algorithm which is quadratic in the coefficient length and which also performs well in the practical test cases.

1. Introduction

A polynomial real root isolation algorithm is an algorithm which, given a polynomial A(x) with real coefficients, computes a sequence of disjoint intervals each containing exactly one real root of A, and together containing all real roots of A. Heindel, in a 1970 Ph.D. thesis, [4], (see [5] for a compact presentation of the main results), provided the first implementation using exact arithmetic within a computer algebra system of such an algorithm, together with a thorough analysis of its computing time. Heindel's algorithm was based on the methodical application of Sturm's theorem.

Just recently, Collins and Loos, [3], have given a similar treatment of a new polynomial real root isolation algorithm, which is based on recursive application of Rolle's theorem together with a certain tangent construction. They show that this new algorithm is significantly superior to the Sturm algorithm in most practical cases.

In the following we present another new algorithm, this one based on the use of Descarte's rule of signs together with certain linear-fractional transformations. We find our algorithm to be a strong competitor of the Collins-Loos algorithm; applied to the same examples which were used in [3], it is from two to eight times faster in most cases. Furthermore, we are able to establish a better theoretical upper bound for its computing time; for a squarefree integral polynomial A of degree n with $|A|_1 = d$, we obtain a bound of $n^6 L(d)^2$ in place of the bound $n^7 L(d)^3$ for the previous algorithms. Here $|\Sigma_{i=0}^{n} a_i x^i|_1 = \Sigma_{i=0}^{n} |a_i|$ and L(d) is the length of the integer d in radix representation with unspecified integer radix $\beta \geq 2$.

The new algorithm to be described in this paper is dedicated to J. V. Uspensky, to whose book, [6], its discovery is directly attributable. Our algorithm is a modification of one which Uspensky (in 1948) claims is most efficient. We quote from page 127 of his book: "The theorem of Rolle and the rule of signs, though they may often help in separating the roots of an equation, do not provide by themselves a complete and exhaustive solution of this important problem. The application of Rolle's theorem for this purpose requires a knowledge of the real roots of the derivative, which is, for the most part, lacking. The rule of signs is a rather weak proposition and, applied to an equation the nature of whose roots is not known, does not give the exact number of positive (or negative) roots except when the number of variations is zero or one. But exactly these two particular cases, when combined with a remarkable theorem published by Vincent in 1836 and hinted at earlier by Fourier, supply the most efficient method, not only to determine the exact number of positive and negative roots but also to effect their separation, in case the proposed equation has no multiple real roots."

We programmed Uspensky's algorithm and found that indeed, on the average, it was faster than the Sturm algorithm. But, as we will show, its maximum computing time is exponential in L(d) and, for the same reason, the observed computing times are occasionally quite unsatisfactory. Our modification of Uspensky's algorithm, which will be described explicitly below, may be regarded an application of the algorithm design principle of balance (see [1], page 65). The modification tends to increase the computing time when the roots are well separated, but decreases the computing time when some roots are very close.

2. Uspensky's Algorithm

Let $b = (b_1,\ldots,b_r)$ be a sequence of real numbers. Let $b' = (b'_1,\ldots,b'_s)$ be the subsequence

*Research supported by National Science Foundation Grant DCR74-13278.

of non-zero elements of b. Then the number of variations in b, var(b), is the number of i's, $1 \le i < r$, such that $\text{sign}(b'_i) = -\text{sign}(b'_{i+1})$. If $A(x) = \Sigma_{i=0}^{n} a_i x^i$ is a real polynomial, then we identify A with its sequence of coefficients $a = (a_0,\ldots,a_n)$ and write var(A). Descartes' rule of signs is a theorem which asserts that the number of positive real roots (multiplicities counted) of a real polynomial A is equal to var(A) - 2k for some non-negative integer k. The exact number of positive real roots is determined just in case var(A) = 0 or var(A) = 1. The theorem can be applied to negative roots by replacing A(x) by A(-x), and zero is a root of A just in case $a_0 = 0$.

If var(A) ≥ 2 and $A(0) = a_0 \neq 0$, let $A_1(x) = A(x+1)$ and $A_2(x) = (x+1)^n A(1/(x+1))$. If $\alpha_1,\ldots,\alpha_m$ are the roots of A greater than or equal to one, then $\alpha_1-1,\ldots,\alpha_m-1$ are the roots of A_1 greater than or equal to zero. If $\alpha_1,\ldots,\alpha_m$ are the roots of A between zero and one, then $\alpha_1^{-1}-1,\ldots,\alpha_m^{-1}-1$ are the positive real roots of A_2. Let $A_1(x) = x^h \bar{A}_1(x)$ where $\bar{A}_1(0) \neq 0$. If h > 0, then 1 is a root of A and {1} = [1,1] is an isolating interval for A. If $\text{var}(\bar{A}_1) = 0$, then A has no roots in (1,∞), while if $\text{var}(\bar{A}_1) = 1$ then (1,∞) is an isolating interval. If $\text{var}(\bar{A}_1) > 1$, then $\bar{A}_1$ is subjected to the same two substitutions, x + 1 and 1/(x+1). Similarly, if $\text{var}(A_2) = 0$, then A has no roots in (0,1); if $\text{var}(A_2) = 1$, then (0,1) is an isolating interval for A; and if $\text{var}(A_2) > 1$, then A_2 is subjected to the same two substitutions.

This is Uspensky's algorithm. Its computations can be represented in the form of a binary tree in which there is associated with each node a polynomial and a transformation. Each path from a node to a successor corresponds to a substitution x + 1 or 1/(x+1), which substitution is applied to both the polynomial and the transformation. Polynomials at terminal nodes have at most one variation; the associated transformations of those with one variation, when applied to (0,∞), produce isolating intervals.

Uspensky applies a theorem of Vincent ([7], 1836) to show that, when applied to a squarefree polynomial, the algorithm always terminates, i.e., the binary tree is finite. Vincent's theorem asserts that if a squarefree polynomial is successively transformed by successive substitutions $a_i + \frac{1}{x}$, where the a_i are positive integers, one eventually obtains a polynomial with at most one variation. In fact, Uspensky's strengthened version of Vincent's theorem bounds the number of substitutions 1/(x+1) as a function of n = deg(A) and the minimum root separation of A.

If m is the number of substitutions 1/(x+1) in any path from the root of the tree to any node, then the Vincent-Uspensky theorem shows that m is dominated by $L(n) + L(\lceil \lambda^{-1} \rceil)$ where $\lambda = \text{sep}(A)$ is the minimum root separation of A. But it is known, [2], that $L(\lceil \lambda^{-1} \rceil) \le nL(d)$ where $d = |A|_1$, so $m \le nL(d)$. However, the number of substitutions x + 1 is not dominated by any power of L(d), as may be verified with the example $A(x) = \{ax-(a-1)\}\{(a+1)x-a\} = (a^2+a)x^2 - (2a^2-1)x + (a^2-a)$, with a a positive integer, a > 2. Here the maximum value of m is a - 2, while $d = 4a^2 - 1$. It is possible to construct a similar example for every n ≥ 2.

In the implementation of Uspensky's algorithm, one may replace the (partially-constructed) binary tree by a list of pairs corresponding to its terminal nodes, each pair consisting of the associated polynomial and transformation. Moreover, the list may contain pairs only for those nodes for which the polynomial has at least two variations. The main loop of the algorithm then removes a pair from the list and produces two new pairs corresponding to the two substitutions. The new pairs with two or more variations are then returned to the pair list while the new pairs with one variation contribute their intervals to an output list. The loop is repeated until the pair list becomes empty. Some observed times for application of this algorithm are presented in Section 4.

3. The Modified Uspensky Algorithm

Table 1 shows the intervals which may occur at the first few levels of the binary tree in Uspensky's algorithm. At level 4, the lengths of the finite intervals fluctuate between 1/15 and 1; at higher levels the disparity in lengths grows rapidly. Clearly there is a lack of optimal balance in this algorithm.

Table 1. Interval's in Uspensky's Algorithm

Level 0:	(0,∞)
Level 1:	(0,1),(1,∞)
Level 2:	(0,1/2),(1/2,1),(1,2),(2,∞)
Level 3:	(0,1/3),(1/3,1/2),(1/2,2/3),(2/3,1), (1,3/2),(3/2,2),(2,3),(3,∞)
Level 4:	(0,1/4),(1/4,1/3),(1/3,2/5),2/5,1/2), (1/2,3/5),(3/5,2/3),(2/3,3/4),(3/4,1), (1,4/3),(4/3,3/2),(3/2,5/3),(5/3,2), (2,5/2),(5/2,3),(3,4),(4,∞)

Let b be a root bound for A, that is, $|\alpha| < b$ for every root α of A. We compute a root bound as in [3], so that $b = 2^k$ for some (positive, negative or zero) ineger k. Let $A^*(x) = A(2^k x)$ if $k \ge 0$, $A^*(x) = 2^{-kn} A(2^k x)$ if k < 0. Now A^* is an integral polynomial whose positive real roots are all in the interval (0,1); the roots of A^* in (0,1) correspond to the positive real roots of A. We associate with A^* the interval (0,b). If $\text{var}(A^*) = 1$, then (0,b) is an isolating interval for A; and if $\text{var}(A^*) = 0$, then A has no positive real roots. If $\text{var}(A^*) \ge 2$, then we put the pair $(A^*,(0,b))$ in a pair list, P. Each pair in P consists of a polynomial A_i and an interval,

(a_i,b_i), such that the roots of A in (a_i,b_i) correspond to the roots of A_i in (0,1) and the intervals (a_i,b_i) are disjoint. The roots of A_i in (0,1) correspond to the positive real roots of $A_i^*(x) = (x+1)^n A_i(1/(x+1))$ and so we assume that $var(A_i^*) \geq 2$ for each i. Let $A_i'(x) = 2^n A_i(x/2)$ and $A_i''(x) = A_i'(x+1) = 2^n A_i(\frac{x+1}{2})$. The roots of A_i' in (0,1) correspond to the roots of A_i in (0,1/2) and hence to the roots of A in (a_i,c_i) where c_i is the midpoint $(a_i+b_i)/2$. Similarly, the roots of A_i'' in (0,1) correspond to those of A in (c_i,b_i). And c_i is a root of A just in case $A_i''(0) = 0$. Thus if $var(A_i'^*) = 1$, then (a_i,c_i) is an isolating interval for A, and if $var(A_i'^*) \geq 2$ then $(A_i',(a_i,c_i)$ is put in the list P. Similarly for $A_i'',A_i''^*$ and (c_i,b_i), but with due regard for the case $A_i''(0) = 0$ as in Uspensky's algorithm. This bisection process is repeated until the list P becomes empty. This is the modified Uspensky algorithm.

Vincent's theorem can be modified to obtain a proof that the modified algorithm terminates. The main result is the following.

Theorem 1. Let A(x) be a real polynomial of degree n. Let $\epsilon_n = (1+1/n)^{1/(n-1)} - 1$. Let α be a real root of A with $0 \leq a < \alpha < b$. Let C be the circle with center (a+b)/2 and radius $(\epsilon_n^{-1}+1/2)(b-a)$. Let $A_1(x) = A((b-a)x+a)$. If all roots, real and complex, of A other than α are outside C, then $var(A_1^*) = 1$.

One also uses the known result that if all roots of a real polynomial A have negative real parts, then var(A) = 0. Combining Theorem 1 with the observation that $\epsilon_n^{-1} = O(n^2)$ and the bound of [2] for minimum root separation, one obtains the following result.

Theorem 2. Let A(x) be an integral square-free polynomial of degree $n \geq 1$ and $|A|_1 = d$. The computing time of the modified Uspensky real root isolation algorithm (using classical arithmetic algorithms) is dominated by $n^6L(d)^2$.

4. Empirical Comparisons

In the following we present several tables showing the observed computing times, for several classes of polynomials, for the Collins-Loos differentiation algorithm (D), the Uspensky algorithm (U), and the modified Uspensky algorithm (M). The polynomials used in these tests are the same as those used in [3], and the times for Algorithm D are taken from [3]. All times are given in seconds and were obtained using SAC-1 on the University of Wisconsin UNIVAC 1110 computer.

We describe the several classes of polynomials very briefly; full descriptions are given in [3]. Tables 2 and 3 pertain to polynomials with random coefficients which are b bits long. n denotes the degree in all tables. The polynomials of Table 4 are resultants of random bivariate polynomials; each entry is the average for three resultants with coefficients from 10 to 30 decimal digits in length. Table 5 is for products of random linear polynomials, with the coefficients of the product truncated to a maximum of 33 bits. Table 6 pertains to the Chebyshev polynomials $T_n(x)$.

Each table has an additional column labeled B. This is the number of bisections performed in Algorithm M. Most of the work of Algorithm M consists of performing translations $A^*(x) = A(x+1)$, and the number of translations performed is 3B. It appears that the average value of B is approximately linear in n, with the constant of proportionality dependent on the class of polynomials. Using this observation (or conjecture) one can argue that the average computing time of Algorithm M is perhaps approximately codominant with $n^4 + n^3L(d)$, which may be compared with the average time $n^4 + n^2L(d)^2$ conjectured for Algorithm D in [3]. In the practical cases observed, both algorithms appear to behave like $n^3L(d)$, but with Algorithm M having a somewhat smaller constant factor.

TABLE 2
Random Polynomials, b=33

n	B	D	U	M
5	2	0.76	0.07	0.17
10	$3\frac{2}{3}$	3.77	0.21	0.66
15	4	8.06	0.32	1.44
20	$4\frac{1}{3}$	18.4	0.68	2.71
25	7	30.2	0.74	7.86

TABLE 3
Random Polynomials, n=15

b	B	D	U	M
33	3	7.9	0.32	0.95
66	5	12.1	0.55	1.89
99	4	10.4	0.43	1.61
132	5	14.2	1.16	2.25

TABLE 4
Random Resultants

n	B	D	U	M
8	7	3.79	0.31	1.01
18	11	21.0	1.76	6.73
32	14	83.4	4.73	33.4

TABLE 5
Random Products

n	B	D	U	M
5	9	1.42	0.14	0.50
10	15	6.83	0.96	2.23
15	32	17.9	4.32	12.5
20	25	38.0	6.77	17.6
25	37	64.0	21.5	40.4

TABLE 6
Chebyshev Polynomials

n	B	D	U	M
5	8	0.78	0.16	0.36
10	14	3.40	1.52	2.02
15	18	8.98	5.36	5.07
20	26	18.6	20.6	17.0
25	30	35.2	47.4	31.6
30	36		118.9	56.0

5. Conclusions

One may be tempted to conclude from the above empirical comparisons that the unmodified Uspensky algorithm is usually faster than the other known algorithms and that therefore these other algorithms do not merit continued use or investigation. Such a simple state of affairs, however, does not exist.

For example, the Collins-Loos algorithm computes isolating intervals not only for A(x) but also for all of its derivatives and, as observed in [3], there are important applications where isolating intervals for several or all derivatives are needed. Another point to observe is that root isolation is often used as a prelude to root refinement or approximation; hence a slower root isolation method which produces smaller isolating intervals may not be as bad as it would appear from the type of data presented above.

The modified Uspensky method is faster than the unmodified Uspensky method in some cases but slower in others. This appears to be another case where the fastest algorithm for "difficult" problems is somewhat slower for easier problems. However, "difficulty" here depends on root distribution as well as degree and coefficient size and since one will generally have little, if any, advance information about root distribution, the selection of the most appropriate algorithm will be difficult. For the present, at least, a computer algebra system should ideally supply several algorithms, allowing the user to exercise judgement in selection of the most appropriate algorithm for the problem at hand. In various contexts the preferred algorithm may be the Collins-Loos algorithm, the Uspensky algorithm or the modified Uspensky algorithm. The Sturm algorithm is generally much slower than the others but, as the data in [3] show, it is slightly faster than the others for Chebyshev polynomials. Certainly there are many questions deserving further study.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.

[2] G. E. Collins and E. Horowitz, The Minimum Root Separation of a Polynomial, Math. Comp., Vol. 28, No. 126 (April 1974), 589-597.

[3] G. E. Collins and R. Loos, Polynomial Real Root Isolation by Differentiation, Proc. 1976 ACM Symposium on Symbolic and Algebraic Calculation, 1976.

[4] L. E. Heindel, Algorithms for Exact Polynomial Root Calculation, Ph.D. Thesis, University of Wisconsin, 1970.

[5] L. E. Heindel, Integer Aritimetic Algorithms for Polynomial Real Zero Determination, Jour. ACM, Vol. 18, No. 4 (Oct. 1971), 533-548.

[6] J. V. Uspensky, Theory of Equations, McGraw-Hill, 1948.

[7] M. Vinent, Sur la Résolution des Équations Numeriques, Jour. de Mathematiques Pures et Appliquees, Vol. 1 (1836), 341-372.

The Aliquot Project:
An Application of Job Chaining in Number Theoretic Computing

M. C. Wunderlich
Northern Illinois University
DeKalb, Illinois 60115

INTRODUCTION. This paper is divided into two parts. Part 1 first presents an old and charming number theoretic conjecture which has been generally believed or at least respected by the mathematical community for over 70 years. A probabilistic argument is then detailed which supports the opposite of this conjecture. A vast amount of computing has been recently done at various universities all over the world in order to investigate the plausibility of this conjecture and it was largely as a result of these computations that the negative argument was formulated. Part II of the paper discusses these computations and describes some of the problems which were encountered in this project. Not only were many hours of machine time consumed but hundreds of man-hours were spent book-keeping and "terminal-watching" because of the non-homogeneous character of the project. Finally, a software system is described which, when fully implemented, will provide the user with controlled but automatic job submission permitting number theoretic projects such as this to be carried out with much less constant attention.

Part I. The mathematics.

An aliquot sequence (abbreviated AS) is a sequence of positive integers $n_0, n_1, \ldots$ for which $n_k = s(n_{k-1})$ where $s(n) = \sigma(n) - n$ and, as usual,

$$\sigma(n) = \sum_{d \mid n} d.$$

We repeatedly use the fact that σ is multiplicative and $\sigma(p^\gamma) = 1 + p + p^2 + \ldots + p^\gamma$. There are three kinds of aliquot sequences:

terminating: $n_k = 1$ for some k

periodic: $n_{k+t} = n_k$ for some t and for k sufficiently large (for example $s(6) = 3 \cdot 4 - 6 = 6$)

infinite: $\lim_{k \to \infty} n_k = \infty$.

Catalan (1887) and Dickson (1913) have conjectured that infinite aliquot sequences do not exist. Recent computations by Guy, Selfridge, and Wunderlich show that of all the aliquot sequences for which $n_0 < 10{,}000$, all but 98 are known to terminate. In January, 1973, Richard Guy and John Selfridge conjectured that infinitely many aliquot sequences, perhaps almost all with n_0 even, are infinite. We present in this section the Guy-Selfridge argument.

Table 1: Example of a "long" terminating AS.

k	n(k)
0	2880
32	123 709593008
55	7447648
69	668429258
99	6677260
154	5108232 531623332
203	26799040
224	177 841798874
251	124124
325	36445367 869087816
393	277
394	1

The sequence beginning with $n_0 = 2880$ demonstrates the behavior of many aliquot sequences. The table above lists all the terms which are relative maxima or relative minima. Table 2 is a detailed look at a segment of the sequence beginning with 1074. The right hand column of the table is the unique factorization of $n(k)$ into primes. (In our notation, exponents are contained in parentheses and a period is used to denote multiplication whenever necessary. Thus 2(3)5.7.977 means $2^3 \cdot 5 \cdot 7 \cdot 977$.) The reader should note that each of the first 25 terms in the table contains a single power of 2 and no 3's whatsoever. On the other hand, all the other small primes appear about the "right" number of times. This pattern seems to coincide with terms which are steadily decreasing in magnitude. On the other hand, the last 19 terms in the segment contain a single power

of 2, at least one power of 3 and a random representation of the other primes. This pattern is associated with terms which are steadily increasing but not as rapidly as the earlier terms are decreasing. We shall fully explain this phenomenon.

TABLE 2 -- A "Snapshot" of AS 1074*

K	N(K)	FACTORIZATION OF N(K)
310	436546562 900403622	2.218273281450201811
311	218273281 450201814	2.17.29.43.1249.9929.415133
312	149021921 791798186	2.22316761.3338789213
313	74510970 979217018	2.1747.21325406691247
314	37319461 709687494	2.2687767.6942465941
315	18659751 690304874	2.47.198507996705371
316	9925399 835268694	2.7(2)31.3267083553413
317	7952081 369012714	2.199.22861.873982063
318	4036505 399288086	2.2018252699644043
319	2018252 699644046	2.89.2503.4529963369
320	1044364 935545554	2.7.71.139.191.39574709
321	793826 522348846	2.7.56701894453489
322	567018 944534914	2.9176053.30896669
323	283509 592485626	2.13.6917.24749.63697
324	174559 350692374	2.331.3491.75532747
325	88145 963899562	2.67.173.293.12977287
326	47282 729665750	2.5(3)19.9954258877
327	45889 133432330	2.5.19.241521754907
328	41058 698334550	2.5(2)13.29.461.4724903
329	44205 596276330	2.5.4420559627633
330	35364 477021082	2.11.1607476228231
331	22504 667195270	2.5.13.173112824579
332	21119 764598890	2.5.139.15194075251
333	17169 305036150	2.5(2)1483.231548281
334	14787 136459234	2.7393568229617
335	7393 568229620	2(2)5.19(2)1024039921
336	8993 118602224	2(4)19.29582626981
337	9348 110126616	2(3)3.152407.2555687
338	14022 327675624	2(3)3.73.241.5647.5881
339	21673 548325656	2(3)3(2)23.107.199.614657
340	40460 999578344	2(3)3(2)561958327477
341	69120 874279866	2.3.67.267601.642533
342	71184 926630022	2.3.83.142941619739
343	72900 226067898	2.3.157.967.4651.17207
344	74020 746222150	2.3.5(2)7.13.2551.2125741
345	152002 020061626	2.3.7(2)977.18713.28279
346	202028 731286214	2.3.7.4810207887767
347	259751 225939514	2.3.79.149.193.19056173
348	272602 050924486	2.3.29.2857.548364877
349	291599 604752154	2.3.48599934125359
350	291599 604752166	2.3(2)17.65003.14659937
351	377374 531293738	2.3(2)11.1905931976231
352	514601 633582838	2.3(2)101.127.44059.50587
353	620322 290876682	2.3(2)41.840545109589
354	756490 598631738	2.3(3)463.30257203369
355	928230 485009862	2.3.14731.18671.562477
356	928459 246951482	2.3.9677.15990824411
357	928651 136960550	2.3.5(2)31.43.59921.77509
358	1504054 894174170	2.3.5.13.31.317.392444389
359	2521407 331366950	2.3.5(2)13.5477.236083513

* Computed by H. J. Godwin, see [6]

Let the set of all positive integers be partitioned into sets S_0, S_1, S_2 and S_3 as follows:

S_0; set of all odd integers

S_1; $n = 2k$ where $(k,6) = 1$

S_2; $n = 2.3.k$ where k is odd

S_3; $n = 2^2.k$.

If $C(n)$ is a condition on n, we will denote with $N_x\{C(n)\}$ the number of positive integers $n \le x$ for which $C(n)$ is true. Thus for $i = 0,1,2,3$, we define the function

$$(1) \qquad B_i(x) = \frac{N_x\{n \in S_i; s(n) \notin S_i\}}{N_x\{n \in S_i\}}$$

which, loosely stated, measures the probability that a term of an aliquot sequence whose order of magnitude is x will "break" out of the set S_i. We will call these functions "break probabilities".

We also define the function A_i to be the average order of the function $s(n)/n$ taken over the set of all $n \in S_i$. Formally, it is defined to be

$$(2) \qquad A_i = \lim_{x\to\infty} \frac{\sum_{\substack{n\le x \\ n\in S_i}} \frac{s(n)}{n}}{\sum_{\substack{n\le x \\ n\in S_i}} 1} = \frac{\sum_{\substack{n\le x \\ n\in S_i}} \frac{\sigma(n)}{n}}{\sum_{\substack{n\le x \\ n\in S_i}} 1} - 1.$$

This function measures the average growth of an aliquot sequence term lying in one of the sets S_i. That the limit exists is based on the following lemma whose proof we omit.

Lemma 1: If k is a residue mod p, then

$$\sum_{\substack{n\le x \\ n\equiv k(p)}} \frac{\sigma(n)}{n} = Cx + O(\log x)$$

for some constant C.

For the remainder of this section, we adopt the usual notation $a \mid n$ for "a divides n". If p is a prime, $p^\alpha \mid n$ but $p^{\alpha+1} \nmid n$, we will write $p \| n$.

Lemma 2:

(a) $B_0(x) = O(1/\sqrt{x})$

(b) $B_1(x) \sim \pi^2/6 \log x \doteq 1.64492/\log x$

(c) $B_2(x) \sim \pi^2/24 \log x \doteq .41123/\log x$

Proof: (a) follows from the fact that if k is odd, $s(k)$ is even if and only if k is a square. ($\sigma(p^a) = 1 + p + p^2 + \ldots + p^a$). To prove (b), we let $(k,6) = 1$ and let $M = s(2k) = 3\sigma(k) - 2k$. Clearly $3 \nmid M$. $2 \| M$ if and only if $2^2 \mid \sigma(k)$ and $2^2 \nmid \sigma(k)$ if and only if

(i) k is a square (neglect this)
(ii) $k = Sp$ where S is a square relatively prime to 6 and p is a prime $\equiv 1 \pmod 4$.

It is well known that the number of primes $\le x$ which are congruent to 1 mod 4 is asymptotic to $x/2\log x$. Using this we obtain

$$(3) \qquad N_x\{n = Sp,\ S \text{ square},\ (S,6) = 1,\ p \text{ prime},\ p \equiv 1(4)\} \sim \frac{\pi^2 x}{18 \log x}$$

so $N_x\{n \in S_1,\ s(n) \notin S_1\} \sim \frac{\pi^2 x}{36 \log x}$ and so from (1), $B_1(x) \sim \frac{\pi^2 x}{36 \log x} \Big/ \frac{x}{6} = \frac{\pi^2}{6 \log x}$. To prove (c), let k be odd and let

$$M = s(2.3^a.k) = 3\sigma(3^a)\sigma(k) - 2.3^a.k.$$

Again we discount the case where k is a square; thus we assume that $2 \mid \sigma(k)$. Therefore $2^2 \nmid M$ whenever $\sigma(3^a)$ is odd and $k = Sp$, S a square and $p \equiv 1 \pmod 4$. $\sigma(3^a)$ is odd whenever a is even, so we get using (3)

$$N_x\{n \in S_2, s(n) \notin S_2\} \sim \frac{\pi^2 x}{18 \log x}\left(\frac{1}{2.3^2} + \frac{1}{2.3^4} + \ldots\right) \sim \frac{\pi^2 x}{288 \log x}$$

and $N_x\{n \in S_2\} \sim x/12$.

Thus $B_2(x) \sim \frac{\pi^2 x}{288 \log x} \Big/ \frac{x}{12} = \frac{\pi^2}{24 \log x}$.

To obtain values for A_0, A_1, and A_2, we need average order results for $\sigma(n)/n$ taken over various sets. General theorems of this sort can be obtained, but we need only two special results whose proofs we can sketch.

Lemma 3:

(a) $$\sum_{\substack{n\le x \\ n \text{ odd}}} \frac{\sigma(n)}{n} \sim \frac{\pi^2 x}{16}$$

(b) $$\sum_{\substack{n\le x \\ (n,6)=1}} \frac{\sigma(n)}{n} \sim \frac{\pi^2 x}{27}$$

Sketch of proof: We use a result of Hardy and Wright to write

(4) $$\frac{\pi^2 x}{6} + O(\log x) = \sum_{n\le x} \frac{\sigma(n)}{n} =$$

$$= \sum_{\substack{k>0 \\ 2^k<x}} \sum_{\substack{n\le x/2^k \\ n \text{ odd}}} \frac{\sigma(2^k n)}{2^k n} .$$

We then use lemma 1 to assert that a C exists for which

$$\sum_{\substack{n\le x \\ n \text{ odd}}} \frac{\sigma(n)}{n} = Cx + O(\log x)$$

and we substitute this for the right hand sum of (4). [Note that $\sigma(n)$ and $\sigma(n)/n$ are multiplicative.] One can thus solve for C and get (a) with the error $O(\log^2 x)$. This result is then used in a similar way to obtain (b).

Lemma 4: $A_0 = \frac{\pi^2}{8} - 1$, $A_1 = \frac{\pi^2}{6} - 1$, and

$$A_2 = \frac{11\pi^2}{48} - 1.$$

Proof: The value of A_0 follows directly from (2) and lemma 3. To obtain A_1, we estimate

$$\sum_{\substack{n\le x \\ n\equiv \pm 2(12)}} \frac{\sigma(n)}{n} = \sum_{\substack{n\le x/2 \\ n\equiv \pm 1(6)}} \frac{\sigma(n)}{n} \cdot \frac{\sigma(2)}{2} \sim$$

$$\sim \frac{3}{2} \sim \frac{\pi^2 x}{2.27} = \frac{\pi^2 x}{36}$$

and the result follows from (2). To estimate A_2, we write

$$\sum_{\substack{n\le x \\ n\equiv 6(2)}} \frac{\sigma(n)}{n} = \frac{3}{2}\left(\sum_{\substack{n<x/2 \\ n \text{ odd}}} \frac{\sigma(n)}{n} - \sum_{\substack{n<x/2 \\ n \pm 1(6)}} \frac{\sigma(n)}{n} \right) \sim$$

$$\sim \frac{11\pi^2}{12.48}$$

and the result follows from (2).

One could also compute $A_3 = \frac{11\pi^2}{48} - 1$, but the result would be misleading. The actual "average growth" of terms n such that $2^2 | n$ would not reflect the average order of $s(n)/n$ since the break probability would differ for each class of integers N_a for which $2^a || n$. The following computation, however, indicates that the general tendency for numbers in S_3 would be to grow upwards. We let $n = 2^a k$ for $a \ge 2$ and k odd and write

$$\frac{\sigma(2^a k)}{2^a k} = \frac{\sigma(2^a)}{2^a} \cdot \frac{\sigma(k)}{k} > \frac{7}{4} \frac{\sigma(k)}{k},$$

but the average order of $\frac{\sigma(k)}{k}$ over k odd is $\frac{\pi^2}{8}$ and

$$\frac{7}{4} \cdot \frac{\pi^2}{8} \doteq 2.15897 > 2.$$

The results thus far can be summarized in the following table:

j	$B_j(x)$	A_n
0	$1/\sqrt{x}$	$\pi^2/8-1$: .23369
1	$\pi^2/6 \log x$	$\pi^2/6-1$: .64493
2	$\pi^2/24 \log x$	$11\pi^2/48-1$: 1.26178
3	≤ 1	> 1

Since for all n, $s(n) \in S_0$ only if n is a square or twice a square, aliquot sequences with large terms will be dominated by terms in S_1, S_2, and S_3. The Guy-Selfridge argument is based on the "expected-value" behavior of an aliquot sequence dominated by terms in S_1 and S_2. Terms in S_3 can only help matters.

Now, suppose ℓ_1 consecutive terms occur of type 1, after which n has been reduced to n'. The geometric mean of the size of the term is $n^{(1+\alpha_1)/2}$. The "mean probability" of this type breaking is

$$\frac{\pi^2}{6 \log n^{(1+\alpha_1)/2}} = \frac{\pi^2}{3(1+\alpha_1)\log n}$$

so the expected length of this string, ℓ_1, is

$$\ell_1 = \frac{3(1+\alpha_1)\log n}{\pi^2} .$$

But, on average, $s(n) = nA_1$ where $A_1 = \frac{\pi^2}{6} - 1$. Thus

$$n\left(\frac{\pi^2}{6} - 1\right)^{\ell_1} = n^{\alpha_1}$$

$$\log n + \frac{3(1+\alpha)\log n}{\pi^2} \log\left(\frac{\pi^2}{6} - 1\right) = \alpha_1 \log n$$

$$\pi^2 + 3(1+\alpha_1)\log\left(\frac{\pi^2}{6} - 1\right) = \pi^2 \alpha_1$$

$$\alpha_1 = \frac{\pi^2 + 3\log\left(\frac{\pi^2}{6} - 1\right)}{\pi^2 - 3\log\left(\frac{\pi^2}{6} - 1\right)} = .76479$$

$$\ell_1 = \frac{3(1+\alpha_1)}{\pi^2} \log n = .53643 \log n.$$

For type 2 sequence,

$$\ell_2 = \frac{12(1+\alpha_2)\log n}{\pi^2}$$

and on average $s(n) = nA_2$ where $A_2 = \frac{11\pi^2}{48} - 1$

so in a string of ℓ_2,

$$n\left(\frac{11\pi^2}{48} - 1\right)^{\ell_2} = n^{\alpha_2}$$

or

$$\log n + \frac{12(1+\alpha_2)\log n}{\pi^2} \log\left(\frac{11\pi^2}{48} - 1\right) =$$

$$= \alpha_2 \log n,$$

$$\pi^2 + 12(1+\alpha_2)\log\left(\frac{11\pi^2}{48} - 1\right) = \pi^2\alpha_2,$$

$$\alpha_2 = \frac{\pi^2 + 12 \log\left(\frac{11\pi^2}{48} - 1\right)}{\pi^2 - 12 \log\left(\frac{11\pi^2}{48} - 1\right)} = 1.78831,$$

$$\ell_2 = \frac{12(1+\alpha_2)}{\pi^2} \log n = 3.39018 \log n.$$

If an aliquot sequence began with terms in S_2, continued for ℓ_1 terms, broke into terms in S_1, continued for ℓ_2 terms, the size would be

$$(n^{\alpha_2})^{\alpha_1} = n^{1.36768}.$$

The total number of terms would be approximately

$$3.39 \log n + .536 \log(n^{1.79}) = 4.35 \log n.$$

Thus the expected behaviour of an aliquot sequence dominated by the type 1 and 2 terms is for geometrical growth. To repeat, type 3 terms can only help, the type 0 "down driver" is entered with prob. $O(1/\sqrt{x})$ and hence can be neglected.

The above argument is an over-simplified analysis of a very complicated two dimensional Markov process. The set of integers should be partitioned into a larger collection of sets, each associated with its own "driver". For a discussion of which drivers should be included, see Guy and Selfridge [7]. A break matrix can be constructed giving the probability of a sequence going from one driver to another - these probabilities are, of course, functions of x, the magnitude of the term. Each driver has its own growth distribution describing how rapidly terms are increasing or decreasing in each category. In order for such a model to be convincing, one should compare it with driver statistics collected from a large number of computed aliquot sequences. We will now turn our attention to the problems associated with the computing of aliquot sequences.

Part II. The Computation. The following algorithm replaces the positive integer N with $\sigma(N) - N$, the next term in the aliquot sequence.

Algorithm A

1. (Initialize) Set $S \leftarrow 1$; set $M \leftarrow N$.
2. Perform steps 2.1 through 2.4 while M is composite.
   - 2.1 Search for p, the smallest prime such that $p|M$. If no such p can be found, the program fails.
   - 2.2 Let $M \leftarrow M/p$ and $F \leftarrow 1 + p$.
   - 2.3 Do while $p|M$; $F \leftarrow 1 + Fp$; $M \leftarrow M/p$; End;
   - 2.4 $S \leftarrow S \cdot F$;
3. If $M > 1$, set $S \leftarrow S \cdot (1 + M)$
4. Let $N \leftarrow S - N$;

For each $p^\gamma || N$, step 2.3 accumulates the value $\sigma(p^\gamma) = 1 + p + p^2 + \ldots + p^\gamma$ and step 2.4 accumulates the product of these values for all $p|N$. In practice, a program would test for $n_k = n_{k-1}$, $n_k = n_{k-2}$ and perhaps even $n_k = n_{k-4}$ since many aliquot sequences are known of period 1, 2 and 4. There is also a well known sequence of period 43 which a sophisticated program can look for. Also, since this program is designed to collect driver statistics, steps 2.1/2.3 must be elaborated upon in order to determine which driver is in effect. These are all easy problems to solve, however, and we omit discussing them in this narrative. The purpose of this paper is to discuss some computational problems which are unique to number theoretic computing and to suggest some novel ways to solve them.

The main difficulty arises in steps 2 and 2.1, the only two places where failure can occur. The condition in step 2 is tested by using Fermat's test for compositeness. If p is a prime, then $a^{p-1} \equiv 1 \mod p$ for any integer a. Thus, if $a^{N-1} \not\equiv 1 \mod N$, N is surely composite. Furthermore, if N is composite, it is very improbable that $a^{N-1} \equiv 1 \mod N$. The exponent a^{N-1} can be computed in $O(\log N)$ operations so we have a very inexpensive method for verifying compositeness. If $a^{N-1} \equiv 1 \mod N$ however, we should really not proceed until we have proved N prime.

Proofs of primality for large numbers N can be long, tedious and fraught with danger. A large collection of theorems related to primality proofs can be read in [1]. A complete algorithm for efficient prime testing appears in [14] and we have a working program which is essentially based on that algorithm. Primality proofs are produced by that program by obtaining partial or complete factorizations of $N - 1$ and $N + 1$. In most cases, when N is less than 40 digits, these factors can be found by the simple divide and factor algorithm [9] which attempts to divide $N \pm 1$ by the primes 2, 3, 5,... until enough factors are collected to obtain a proof. The entire proof usually takes less than 12 seconds on a 360/67 computer and so the program can be easily invoked as a sub-procedure of the main aliquot generating program. However, there are cases in which $N + 1$ and $N - 1$ only have a few small divisors--insufficient for a proof. If after

dividing out these factors, one or more of the resulting cofactors fails Fermat's test for compositeness, then the program continues recursively by attempting a primality proof of the cofactor. If both cofactors are composite, a more sophisticated proof can often be obtained by using not only the factors of $N \pm 1$ but also a bound b for which $p|N \pm 1 \Rightarrow p > b$. The program can compute the smallest such bound b which would be sufficient to provide a proof if no additional factors were discovered. If this value is too high, the routine must send back to the main program the message, "We're sorry. Your number is most assuredly prime but we cannot produce a proof of such without possibly investing n minutes of computer time," Where n may be anywhere from one or two to several hundred.

The main program now has basically two options. It can proceed with its computation after noting that the number in question has been merely proved "pseudoprime." Then at the end of the semester when more computer time is available or when the economy improves and a new computer is obtained at our university these difficult primality proofs can be "cleaned up" and the PSP's denoting pseudoprimes can be removed from the aliquot sequence output. Of course, there is that minute possibility that a pseudoprime may turn out to be composite and all the subsequent terms in the aliquot sequence would be incorrect. Thus the other more conservative option would be for the main program to halt all operations until a complete primality proof is obtained. We adopted the former philosophy in our computations. In fact, we employed a very crude primality proving program for the bulk of the aliquot project while we were developing the more sophisticated program described above. By now, all PSP's have been removed from our output and not a single composite pseudomprime was discovered.

The other difficulty arises in step 2.1 of the algorithm. We have already assertained that M is composite but all of our efforts to find a factor of M have proved unsuccessful and we are faced with the possibility of exceeding our estimated computer time for the job. At this point, there is really nothing the main program can do but terminate the job. There is no such thing as a pseudo-factorization which would enable the program to carry on. Since both difficulites in this algorithm involve the factorization of large numbers, we shall digress for a moment to discuss the general question, "How long does it take to factor a number."

It depends very strongly on the method you wish to use, and having chosen the method it may depend on the number you are factoring. In order to completely factor n using the divide and factor method discussed earlier, one must divide by the primes which are less than the second largest prime dividing n, which we will denote by $F_2(n)$. D. E. Knuth and L. T. Pardo [9, 10] have recently analysed the distribution of the values $F_2(n)$ for $n \leq x$ and their results show that for about half of the integers $n \leq x$ $F_2(n) \leq x^{.21172}$. Thus, for about half the 36 digit numbers we factor, a complete factorization can be gotten by dividing up to 42 million. On our computer, this would take about 7.3 minutes of computer time. To put it another way, we can factor a 36 digit number up to 1,000,000 in about 10 seconds. Again, using the Knuth-Pardo tables, we completely factor about 37% of the 36 digit numbers by dividing up to 1,000,000. However, for 25% of the 36 digit numbers, $F_2(n) > n^{.29153} \doteq 31 \times 10^9$. To factor this high would take about 90 hours of machine time. There are two other recently discovered methods of factoring which shorten this time considerably. Pollard's Monte Carlo method requires roughly $\sqrt{p}$ operations where p is the second largest prime dividing n, but each operation is at least 100 times as costly as the single division counted in the divide and factor algorithm. Never-the-less, in the example cited just above, only 176823 operations would be required. This should be possible in a few minutes with a good program. Even with the Pollard Monte Carlo method, one 36 digit number in 20 will take over 4 hours of computer time. (Another interesting method was discovered earlier by Pollard which requires p operations where p is the largest prime dividing $q - 1$ where $q|n$. [12,13]) For two good discussions of factorization, see [8,10].

There are other methods of factoring which do not depend at all on the particular distribution of the factors but rather depend only on the size of the number. The best example of this type of factorizer is the continued fraction method developed by John Brillhart [11]. Although performance characteristics have not been theoretically obtained for this method, our experience with the program suggests the timing formula

$$\text{TIME} = .0003324\ N^{.1574} \tag{4}$$

where TIME is the estimated time in 360/67 computer minutes to factor the number N. This formula yields the following values.

N	TIME (minutes)
10^{16}	.110
10^{20}	.467
10^{24}	1.992
10^{28}	8.489
10^{32}	36.18
10^{36}	154.2

The formula was obtained by doing a linear regression analysis of log T versus log N on 100 actual factorizations using the program in which N was factored in T minutes. The 100 observations ranged in size from 10^{16} to 10^{36}. The fit was quite good producing a correlation coefficient of 0.968. For each of these observations the ratio ACTUAL TIME/PREDICTED TIME was computed. The largest and second largest of these ratios

were 2.84 and 2.01 respectively and the smallest ratio was .492. The computed mean was 1.05 and the standard deviation was .344. Thus the timing estimate will almost always predict the actual time to within a factor of 2.

The point we are making is that any general factoring subroutine presents the calling program with a hopeless problem. There is absolutely no way the main program can know how much computer time it will take to factor a number. It may take less than 10 seconds or over one hour. Clearly the 10 second jobs can be handled as a normal subroutine but what happens when a main program with a 5 minute time estimate calls a factoring subprogram which decides it requires over an hour to do its task? It must send back the message, "We cannot factor your number in the alloted time. You had better terminate your job."

Thus, the computation of a single sequence up to the limits of our computational power was a tedious and time consuming project. The early elements of the sequence were computed very rapidly, but as soon as hard-to-factor number was encountered, the program simply "timed out" and the output obtained was filed away. The difficult number was then submitted to a variety of programs for factorization and ultimately, if no other method succeeded, it was submitted to the continued fraction program. This usually required an over-night run so that in the morning after we collected our factor from the output bin, we had to compute the next term of the aliquot sequence on a desk computer (the Ollivetti 101) before we could rekindle our aliquot program. (In the later stages of the project, our aliquot program could receive a "hint list" of large factors which it would always try before giving up. This feature also provided us with a relatively fast procedure for recalculating a long sequence in order to provide contiguous output.) Since we had a very large number of sequences to compute, we generally had four or five going at any one time. The bookkeeping was very tedious and the probability of error was disturbingly high. At one point in the project, we were all very excited at the prospect of the sequence beginning with 4488 exceeding 1000 terms in length. When we made a recalculation of the sequence using our Hint feature, we discovered that many months (and terms) earlier a mistake was made on the desk computer and we had been computing a different and thoroughly uninteresting sequence ever since. In fact, 4488 terminates with the 459th term. Mistakes such as these were very rare--a credit to precision bookkeeping and data handling, not to versatile and efficient software.

Our new FACTOR program, which is in the design stage at the time of this writing (March, 1976), will automate this entire process. It's novelity relies on the fact that under the IBM OS operating system using HASP any program can send data to the internal reader, a special output channel provided by the system. Such data is immediately processed by HASP and introduced into the input stream as a job which will be queued into execution along with any other jobs that are currently awaiting execution. Thus a program has the capability of submitting another program for execution into the job queue. The program FACTOR has five parameters and could be invoked from an assembly language program using the following macro call.

CALL FACTOR,(NUMBER, TIME, AFACT, #FACT, DSN)

The parameters are used as follows:

NUMBER - The number which is to be factored

TIME - The maximum amount of computer time which FACTOR can use in factoring the number.

AFACT - A pointer to the address in the main program where the factors are to be stored.

#FACT - On input, this is the maximum number of factors which can be stored at AFACT. After execution, it contains the number of factors.

DSN - A pointer to a character string which is the name of a catalogued data set located on an I/O device such as a magnetic disk.

The program also utilizes an external file called MEMORY. Its use will be described in detail later but for the moment, it will suffice to say that whenever a "major effort" is required to obtain a factorization, such as the use of Brillhart's continued fraction program, the number and its factorization is placed on the file MEMORY. We now describe the operation of FACTOR.

1. (Has the number already been factored?) The program first searches the file MEMORY to see if the number and its factorization has already been obtained. If so, it returns the factors to the main program, deletes them from the MEMORY file and returns.

2. (The number is not on MEMORY) The program attempts to factor the number in the time alloted. It can succeed in two ways.

a) It obtains a complete factorizations and it has factored sufficiently far to guarantee that all the factors are indeed prime including the largest one. In this case, it returns the factors and returns to the main program with a condition code of 10.

b) It obtains a complete factorization but the largest factor has only been tested for being pseudoprime. That is, it failed Fermat's test for compositeness. In this case, the factors are returned and the program returns after setting the condition code to 4. Thus we make it the responsibility of the main program to do the prime testing.

3. (The program fails) The TIME allotted by the main program wasn't sufficient to factor the number. In this event, the factors obtained together with the composite cofactor are returned and preparation is made to return to the main

program with a condition code of 8. First, however, an estimate is made as to how much additional computer time may be required to complete the factorization and if this amount of time is not excessive a job is introduced into the job stream by writing to the internal reader. This job, which we will call FACTOR2, will not execute immediately, of course, but will ultimately be queued into execution by the operating system.

4. (FACTOR2 executes). This program can now use all the high powered and time consuming methods it needs in order to factor the troublesome number. If it succeeds it performs the following two operations before terminating.

a) The factorization together with the original small factors (if any) found by the initial execution of the program are placed on the MEMORY file so that subsequent execution of the main program will obtain the factorization immediately when it re-invokes the FACTOR program.

b) The data set whose name is DSN is written to the internal reader as an executable job. This is the data set name originally communicated to FACTOR by the original call statement in the main program. If the main program is contained in the data set DSN, control is effectively passed back to the main program through the job submission process.

If the factoring is not successful, the procedure is essentially the same. The partial factorization is put on the MEMORY file and it is marked as being essentially unfactorable. Then DSN is submitted as described as above. When FACTOR is again invoked, the partial factorization can be returned with a condition code of 12, telling the main program that a complete factorization is not feasible with present technology.

It is now evident how a main program can be designed around this factorization system to generate an aliquot sequence. The input for the program would reside on a data set. Whenever the program terminates either by exceeding execution time or attempting to factor a difficult number, restart information is written to the input data set. If the aliquot program exceeds time itself, it must first submit the data set DSN to the internal readers so that the program will re-execute. The output generated by the program must be directed to an output data set rather than being sent to the system printer. Thus a clean copy of the output can be obtained after the entire chain of jobs is completed.

This has been an oversimplified description of the factorization system which we really want to implement. We will conclude this paper by briefly describing some additional features which this system will ultimately have.

a) An additional parameter BOUND should be provided. This tells the factor program that although a complete factorization would be nice, it would be acceptable to provide a set of factors the largest of which has no factor less than BOUND. This would be important for prime testing. On output, in case of failure, BOUND could be set by FACTOR to the extent to which factoring has been done.

b) The factoring program will automatically collect performance statistics and store them on another auxiliary file. It is very important that we continually sharpen our time and core estimates regarding factoring.

c) A parameter MAXTIME could tell FACTOR how large a job it is permitted to submit to the external reader. The user may decide that certain number theoretic projects aren't important enough to consume hours of computer time every night.

d) The main program may wish to decide after receiving partial factorization information whether or not FACTOR2 should be submitted. There are a variety of ways to implement this feature. A special macro PAUSE can be provided which calls an alternate entry point in FACTOR which submits the job FACTOR2 and then returns normally. If the main program ends with a normal RETURN, the job will not be submitted.

e) The program FACTOR2 will itself be a chain of separate jobs. We already have a very successful version of Brillhart's continued fraction program which submits a sequence of small jobs each of which generates a collection of factored quadratic residues. When a large enough collection of factored residues are generated, a 1 minute program is submitted which completes the factorization process by doing a Gaussian elimination on a very large (360,000 bytes) 0-1 matrix. One of our 36 digit factorizations took from Wednesday afternoon to Saturday morning to complete its work, going through a chain of 10 twenty minute jobs.

f) When this system becomes heavily used, a queuing system will be designed. A file FQUEUE will be maintained which contains all the numbers which are awaiting factorization. The file will be sorted according to the size of the number so that small jobs will be executed first. When the in-stream FACTOR program submits FACTOR2, this merely inserts the number together with any small factors which have been found and the return DSN onto the file FQUEUE and sorts it. A separate chain of factoring programs will be removing numbers from the top of the file and factoring them as time permits. Provision will also be made for entering numbers onto FQUEUE from a computer terminal. In this case, the character string DSN will serve as a label rather than a return program so that the factored number can be identified.

REFERENCES

1. J. BRILLHART, D. H. LEHMER, J. L. SELFRIDGE, "Primality testing and new factorizations of $2^n \pm 1$," Math. Comp. 29 (1975).

2. E. CATALAN, Bull. Soc. Math. France, v. 16, 1887-88 pp. 128-129.

3. L. E. DICKSON, "Theorems and tables on the sums of divisors of a number," Quart. J. Math. v. 44, 1913, pp. 264-296.

4. R. K. GUY and J. L. SELFRIDGE, "Interim report on aliquot series," Proc. Manitoba Conf. Numerical Math., Winnipeg, 1971, pp. 557-580.

5. R. K. GUY, D. H. LEHMER, J. L. SELFRIDGE and M. C. WUNDERLICH, "Second report on a aliquot sequences," Proc. 3rd Manitoba Conf. Numerical Math., Winnipeg, 1973, pp. 57-368.

6. R. K. GUY and J. L. SELFRIDGE, Combined Report on Aliquot Sequences, University of Calgary Math. Research Report No. 225, May 1974.

7. R. K. GUY and J. L. SELFRIDGE, "What drives an aliquot sequence," Math. of Comp. v. 29, n. 129, 1975, pp. 101-107.

8. R. K. GUY, "How to factor a number," Proc. Fifth Manitoba Conf. on Numerical Math. 1975.

9. D. E. KNUTH and L. T. PARDO, "Analysis of a simple factorization algorithm", Submitted for Publication.

10. D. E. KNUTH, "The art of computer programming," Vol. II, Revised edition. Addison-Wesley, to appear.

11. M. A. MORRISON and J. BRILLHART, "A method of factoring and the factorization of F_7," Math. of Comp. v. 29, n. 129, 1975, pp. 183-206.

12. J. M. POLLARD, "Theorems on factorization and primality testing," Proc. Cambridge Philos. Soc. 76 (1974) 521-528.

13. _____, "A Monte Carlo method for factorization," Nordisk Tidskr. Information-sbehandling (BIT) 15 (1975).

14. J. L. SELFRIDGE and M. C. WUNDERLICH, "An efficient algorithm for testing large numbers for primality," Proc. Fourth Manitoba Conf. on Numerical Mathematics, 1974, pp. 109-120.

Generalization of Manin's Conditional Algorithm

by Horst G. Zimmer

Fachbereich 9 Mathematik
Universität des Saarlandes
D-6600 Saarbrücken, Fed. Rep. Germany

1. Introduction and Manin's theorem

In a lecture [16] delivered at the conference on "Utilisation des calculateurs en mathématiques pures" in France, we developed some ideas regarding the implementation of Manin's conditional algorithm [7] for computing the rank and a basis of the group of rational points on an elliptic curve C over the rationals $\mathbb{Q}$. The aim of the present paper is to discuss to which extent this algorithm can be generalized to elliptic curves C defined over an arbitrary finite-algebraic number field K. Furthermore, certain problems concerning the implementation of the generalized algorithm will be treated. Some results obtained in this connection seem to be also per se of interest. In an appendix we shall give complete proofs of the assertions made in sections 1,4, and 6.

Let C denote an elliptic curve in Weierstrass normal form

$$Y^2 = X^3 + aX + b$$

with coefficients a,b in an algebraic number field K of finite degree $n=[K:\mathbb{Q}]$ over $\mathbb{Q}$. The nonzero discriminant of C is $\Delta = 4a^3+27b^2$. Without loss of generality, the coefficients a,b and thus the discriminant Δ may be assumed to be integers in K. By the Mordell-Weil theorem, the additive abelian group C_K of rational points of C over K is finitely generated. Hence this group can be written as a direct sum

$$C_K = \tilde{C}_K \oplus \hat{C}_K$$

of the finite subgroup $\tilde{C}_K$ of all torsion points (points of finite order) in C_K and a maximal free subgroup $\hat{C}_K$ of C_K of finite rank. The number r of basis elements of $\hat{C}_K$ is called the rank of the elliptic curve C over K. Manin's conditional algorithm consists in determining first the torsion subgroup $\tilde{C}_K$ of C_K and second a maximal free subgroup

$$\hat{C}_K \cong C_K/\tilde{C}_K \text{ of } C_K$$

in terms of a basis. Of course, this yields at the same time the rank r of C over K. The conditions on which the algorithm depends are the truth of the conjecture of Birch and Swinnerton-Dyer [7,1o] and the feasibility of estimating the L-series L(C,s) of C over K and its derivatives $L^{(i)}(C,s)$ at s=1, where i ranges from 1 through a certain positive integer r' (see [7] and Manin's theorem below). The latter condition is satisfied in the case $K=\mathbb{Q}$ provided that the Weil conjecture is true (see [7,11]).

Before stating Manin's theorem on which the algorithm is based we have to introduce some notation and recall certain facts.

Let us begin with defining the Néron-Tate height $\hat{h}$ on C_K which is an important tool in the algorithm. To this end we shall use an auxiliary function d on

C_K. The function d is the natural generalization of a corresponding function introduced in [16] for defining $\hat{h}$ in the special case K=ℚ. For a point P= (ξ,η) in C_K, the (affine) coordinates $\xi,\eta \in K$ can be written in the shape

$$\xi = \frac{x}{z^2} \text{ and } \eta = \frac{y}{z^3} ,$$

where x,y, and z are some integers in K. If ξ and η are both nonzero the integers x,y, and z are all nonzero and can moreover be chosen in such a way that the g.c.d.'s of x,z^2 and y,z^3 (as divisors of K) are

$$(x,z^2)=c^2 \text{ and } (y,z^3)=c^3$$

with an integral divisor c of K belonging to a fixed system V of representatives of the (finitely many) divisor classes of K. The divisors c will in fact be chosen in such a way that their norms are bounded by the Minkowski bound B (see [4,12,13]and appendix for details).

In what follows log will always denote the logarithm to a fixed base.

We now define the function d on C_K by setting

$$d(P)=\frac{3}{2n} \log \prod_{\nu=1}^{n} \max \{ \sqrt[2]{|a^{(\nu)}|}\ |z^{(\nu)}|^2 ,$$

$$\sqrt[3]{|b^{(\nu)}|}\ |z^{(\nu)}|^2,\ |x^{(\nu)}| \} \text{ for } P\in C_K ,$$

where $a^{(\nu)}$, $b^{(\nu)}$, $x^{(\nu)}$, and $z^{(\nu)}$ designate the images of a,b,x, and z under the n = [K:ℚ] distinct embeddings

$$K \xrightarrow{\cong} K^{(\nu)} \subseteq \mathbb{C} \quad (\nu=1,2,\ldots,n)$$

of the field K into the complex numbers ℂ and |...| denotes ordinary absolute value on ℂ. This definition can also be used if ξ=0 by simply choosing x=0, z=1. If P=0=(∞,∞) is the neutral element of the addition in C_K, we set d(P)=0. Notice that the function d arises from the usual Weil height h on C_K by only a slight modification [3,8,14]. We call d therefore the <u>modified Weil height</u> on C_K. The <u>Néron-Tate height</u> $\hat{h}$ on C_K is now defined by putting

$$\hat{h}(P) = \lim_{m\to\infty} \frac{d(2^m P)}{2^{2m}} \text{ for } P\in C_K.$$

The function $\hat{h}$ gives rise to a symmetric bilinear form g on $C_K \times C_K$ via [8]

$$g(P,Q)=\frac{1}{2} \{\hat{h}(P+Q)-\hat{h}(P)-\hat{h}(Q)\} \text{ for } (P,Q) \in C_K \times C_K.$$

By means of the bilinear form g we define the <u>determinant</u> of the elliptic curve C over K as the quantity

$$H=|\det(g(P_i,P_j))|_{i,j=1,\ldots,r} ,$$

where the points $P_1,\ldots P_r$ are a basis of a maximal free subgroup $\hat{C}_K$ of C_K. The determinant H clearly does not depend on the choice of the group $\hat{C}_K$ and the points $P_1,\ldots,P_r$ in $\hat{C}_K$ since any other choice just amounts to applying a unimodular transformation of determinant ±1 to the points $P_1,\ldots,P_r$ originally chosen.

We need furthermore a lower bound h' for the values of the Néron-Tate height $\hat{h}$ on the set $C_K \setminus \tilde{C}_K$ (the complementary set to $\tilde{C}_K$ in C_K), that is,a real number h' such that

$$0<h' \le \min \{\hat{h}(P);\ P\in C_K \setminus \tilde{C}_K\} .$$

Finally, the deviation of the modified Weil height d from the Néron-Tate height $\hat{h}$ on C_K plays a significant role in Manin's theorem.

Let δ denote an upper bound for this deviation such that we have

$$|d(P)-\hat{h}(P)| \le \delta \quad \text{for } P\in C_K .$$

It is for the sake of keeping the bound δ as small as possible and making the function $\hat{h}$ easily computable that we have chosen the above modified Weil height d instead of the usual Weil height h for defining the Néron-Tate height $\hat{h}$ on C_K.

Manin's theorem assumes exactly the same shape as in the special case K=ℚ (see [7,16]), though the constants involved become somewhat more complicated.

Theorem (Manin [7,16]).

Suppose that $r' \in \mathbb{Z}$ and $H' \in \mathbb{R}$ are upper bounds for the rank r and the determinant H of the elliptic curve C over the number field K. Then, the set of points $P \in C_K \setminus \tilde{C}_K$ satisfying the inequality

$$d(P) \leq \delta + \frac{2^{2r'}}{c_{r'}^2} H'^2 \max\{1, h'^{2(1-r')}\},$$

where $c_{r'}$ stands for the volume ot the r'-dimensional unit ball in Euclidean space, generates a subgroup of index $\rho \leq r'!$ in a certain maximal free subgroup $\hat{C}_K \cong C_K/\tilde{C}_K$ of C_K.

Of course, the set of points $P \in C_K \setminus \tilde{C}_K$ mentioned in the theorem is finite since there are only finitely many points in C_K having bounded Weil height [8,14].

The proof of the theorem (see [7]) is based on the method of successive minima from geometry of numbers applied to the lattice generated by the $\mathbb{Z}$-module C_K in the r-dimensional real space $C_K \otimes_{\mathbb{Z}} \mathbb{R}$. The Néron-Tate height $\hat{h}$ is a positive definite quadratic form [7,8,14] on $C_K \otimes_{\mathbb{Z}} \mathbb{R}$ and thus defines a Euclidean norm on this space. Since the kernel of the canoncical $\mathbb{Z}$-module homomorphism $C_K \to C_K \otimes_{\mathbb{Z}} \mathbb{R}$ is precisely the torsion subgroup $\tilde{C}_K$ of C_K, we get an injective embedding of any maximal free subgroup $\hat{C}_K \cong C_K/\tilde{C}_K \hookrightarrow C_K \otimes_{\mathbb{Z}} \mathbb{R}$ and may therefore identify $\hat{C}_K$ with the lattice generated by C_K in the space $C_K \otimes_{\mathbb{Z}} \mathbb{R}$.

Manin's algorithm now consists mainly in finding, by trial and error, all points satisfying the inequality of the theorem and then deriving from this set of points a maximal free subgroup $\hat{C}_K$ of C_K in terms of a basis. This is achieved by repeated "divisions by two" and the "infinite descent" procedure known from the proof of the Mordell-Weil theorem [8]. So far the algorithm was restricted to the case $K=\mathbb{Q}$. As pointed out at the beginning, we wish to discuss here the possibilities of extending it to the general case of an arbitrary finite-algebraic number field K as ground field for the curve C. The intended discussion concerns the determination of the torsion subgroup $\tilde{C}_K$ of C_K and the computation of the constants appearing in Manin's theorem.

2. The torsion subgroup

Manin (see [9]) has shown that, for any fixed prime p, the order of the p-component of the torsion subgroup $\tilde{C}_K$ of the rational point group C_K of any elliptic curve C over K is bounded by a constant depending only on K and p. On the basis of a similar result Dem'janenko [1] established the boundedness of the order of the group $\tilde{C}_K$ itself, the bound depending only on the field K (though the effectiveness of Dem'janenko's arguments does not seem to be quite clear). Thus, in some sense, there are not too many torsion points in C_K. The actual determination of the group $\tilde{C}_K$ can be accomplished by applying the height functions d and $\hat{h}$ on C_K in combination with a generalization of the classical theorem of Nagell and Lutz. This theorem holds originally only in the special case $K=\mathbb{Q}$ but it can be generalized to an arbitrary finite-algebraic number field K as was shown in [15], theorem 2. Similarly, the strengthened version of the theorem of Nagell and Lutz used in [16] for computing $\tilde{C}_K$ if $K=\mathbb{Q}$ carries over to the general case of an arbitrary finite-algebraic number field K (see theorem 1 in [15]). Other methods have been recently developed in [2].

We shall state the generalized strengthened version of the Nagell-Lutz theorem in a form which is most appropriate for computational purposes. To this end we introduce the (formal) coefficient divisor [15]

$$m = \prod_{\mathfrak{p}} \mathfrak{p}^{\bar{\mu}_{\mathfrak{p}}}$$

of the Weierstrass equation for C by

defining, for each prime divisor (finite place) $\mathfrak{p}$ of K, a rational exponent

$$\bar{\mu}_{\mathfrak{p}} = \min\{\tfrac{1}{2} w_{\mathfrak{p}}(a), \tfrac{1}{3} w_{\mathfrak{p}}(b)\} ,$$

where $w_{\mathfrak{p}}$ denotes the normalized valuation on K belonging to $\mathfrak{p}$ such that $w_{\mathfrak{p}}(K^*)=\mathbb{Z}$. A divisibility relation $\mathfrak{m}|\gamma$ between the (formal) divisor $\mathfrak{m}$ of K and a nonzero element $\gamma\in K$ is to be understood in the usual sense that $\bar{\mu}_{\mathfrak{p}} \le w_{\mathfrak{p}}(\gamma)$ for all prime divisors $\mathfrak{p}$ of K.

Theorem.

Let $P=(\xi,\eta)$ be a point of order $\mathfrak{m}$ in $\tilde{C}_K$. Then, the following divisibility relations hold:

$$\mathfrak{m}|\xi \ , \ \mathfrak{m}^3|\eta^2$$

if $\mathfrak{m}$ is not an odd prime power, and

$$\mathfrak{m}\mathfrak{p}^{-\frac{2}{p-1}} |\xi \ , \ \mathfrak{m}^3\mathfrak{p}^{-\frac{6}{p-1}} |\eta^2$$

if $\mathfrak{m}=\mathfrak{p}^{\nu}$ $(\nu\ge 1)$ is an odd prime power, and furthermore,

$$\eta^2|\Delta\mathfrak{m}^{-3}$$

if $\mathfrak{m}$ is neither 2 nor twice an odd prime power, and

$$\eta=0 \quad \text{or} \quad \eta^2|\Delta\mathfrak{m}^{-3}\mathfrak{p}^{\frac{2}{p-1}}$$

if $\mathfrak{m}=2$ or $\mathfrak{m}=2\mathfrak{p}^{\nu}$ $(\nu\ge 1)$ is twice an odd prime power.

The proof of this theorem is to be found in [15].
The theorem, giving only a necessary condition for a point $P=(\xi,\eta)$ in C_K to be a torsion point, can be applied in combination with the height functions d and $\hat{h}$ on C_K in order to determine the finite group $\tilde{C}_K$ as follows. A point P in C_K belongs to $\tilde{C}_K$ if and only if $\hat{h}(P)=0$ (see [14]). Therefore, we conclude from section 1 that $\tilde{C}_K$ is contained in the finite set $\mathfrak{S}_K$ of all points P in C_K satisfying

$$d(P)\le\delta$$

with the real bound δ which will be explicitly given in section 4. The set $\mathfrak{S}_K$ is known from Manin's theorem. Now we determine the subset $\mathfrak{T}_K\subseteq\mathfrak{S}_K$ consisting of the zero point O and all points $P=(\xi,\eta)$ in $\mathfrak{S}_K$ which fulfill the necessary condition of the above theorem. Then we get rid of those points in $\mathfrak{T}_K$ having infinite order by checking repeatedly if $2P\in\mathfrak{T}_K$ whenever $P\in\mathfrak{T}_K$. In case $2P\notin\mathfrak{T}_K$ for some $P\in\mathfrak{T}_K$ we discard the point P from the set $\mathfrak{T}_K$. After a finite number of steps we end up with the searched group $\tilde{C}_K$.

3. Bounds for the rank

Up to now it is not known if the rank r of any elliptic curve C over a number field K is bounded by a constant depending only on K or if r can become arbitrarily large as C ranges say over a suitably chosen infinite set of elliptic curves C over K. However, for a fixed elliptic curve C over K upper bounds for r, depending on C and K, can be given.

Whereas in the special case $K=\mathbb{Q}$ treated in [16] we used a method of Tate for estimating the rank r of C over K, we shall here refer to a bound for r which was recently given by Heuss [5] in the general case of a finite-algebraic number field K as ground field for C. Instead of the assumption, made in [16], that the group $C_{\mathbb{Q}}$ contains a point of order 2, we shall here require the group C_K to contain all points of order l of the curve C for a fixed prime number l. These so-called l-division points of C are known to form a subgroup of isomorphism type $\mathbb{Z}/l \times \mathbb{Z}/l$ in the torsion group $\tilde{C}_{\bar{K}}$ of the curve C over the algebraic closure $\bar{K}$ of K. The requirement imposed on the rational point group C_K amounts to a rather severe restriction of generality. Of course, one can always shift from K to

the finite extension T of K obtained by adjoining to K the coordinates of the l-division points of C and thus shift from C_K to the rational point group C_T of C considered as an elliptic curve over T. But then one encounters the open problem as to how the rank of the curve C behaves under the shift from K to the l-division field T over K as the new ground field for C.

Now we shall state the theorem giving a bound for the rank r of the elliptic curve C over K provided that C_K contains the l-division points of C for a fixed prime number l. Let S denote the set of prime divisors $\mathfrak{p}$ of K such that either $\mathfrak{p}$ divides the prime number l or the function field F_C/K of the curve C over K has bad reduction modulo $\mathfrak{p}$. Observe that S is a finite set.

Theorem (Heuss [5].

Let C be an elliptic curve over a finite-algebraic number field K whose rational point group C_K contains the l-division points of C for a fixed prime number l. Denote by S the set defined above. Then, the rank r of C over K satisfies the inequality

$$r \leq 2\{rk_l \mathfrak{C}_S(K) + rk\, u(K) + |S|\} + 1,$$

where $rk_l \mathfrak{C}_S(K)$ and $rk\, u(K)$ stand for the usual rank of the l-component of the S-class group $\mathfrak{C}_S(K)$ of K and the usual rank of the unit group u(K) of K, respectively, and |S| designates the cardinality of the set S.

Remark. Section 2 yields right away the decision as to whether or not the condition of the theorem regarding the l-division points of C is satisfied for the given elliptic curve C over the algebraic number field K. As G. Frey pointed out to the author, a similar estimate for r can be derived from the theorem of Bashmakov and Tate (see [7]) even without the restriction made in the above theorem.

It is the bound of this theorem which we propose as the choice of the number r' occurring in Manin's theorem. This bound proves to be sharper than a similar bound exhibited by Honda (see [5,6]).

4. Calculations involving the Néron-Tate height

Let us first estimate the deviation of the function d defined in section 1 from the Néron-Tate height $\hat{h}$ on the rational point group C_K. Recall that K is a finite-algebraic number field of degree $n=[K:\mathbb{Q}]$. We have $n=r_1+2r_2$, where r_1 is the number of real and $2r_2$ the number of complex embeddings of the field K into the complex numbers $\mathbb{C}$. Designate by D the discriminant of the field K. Then we have for $P \in C_K$

$$-2 \log 2 \leq d(P)-\hat{h}(P) \leq 5 \log 2+\frac{9}{2n}\mu_\infty+\frac{3}{n}\log B,$$

where the quantity μ_∞ is defined by (compare section 1 and appendix)

$$\mu_\infty = \log \prod_{\nu=1}^{n} \max\{\sqrt[2]{|a^{(\nu)}|}, \sqrt[3]{|b^{(\nu)}|}\}$$

and B denotes the Minkowski bound of K (see [4,12,13]), that is,

$$B = \left(\frac{4}{\pi}\right)^{r_2} \frac{n!}{n^n}\sqrt{|D|}\,.$$

The above estimate for the difference $d(P)-\hat{h}(P)$, where $P \in C_K$, coincides precisely with the one indicated in [16] in the special case $K=\mathbb{Q}$ because we have then n=1 and the term involving the Minkowski bound disappears. Accordingly, we choose the constant δ of Manin's theorem as

$$\delta = 5 \log 2+\frac{9}{2n}\mu_\infty+\frac{3}{n}\log B$$

which is the adaquate generalization of the choice of δ made in [16].

The determination of a lower bound h' for the Néron-Tate height $\hat{h}$ on $C_K \setminus \tilde{C}_K$ is achieved as in [7,16]. Since

$$d(P)-\hat{h}(P) \leq \delta \quad \text{for } P \in C_K\,,$$

it suffices to choose

$h' = \min \{\delta, \hat{h}(P);\ P \in C_K \setminus \tilde{C}_K$ such that $d(P) < 2\delta\}$

because there holds $\hat{h}(P) \geq \delta$ for all $P \in C_K$ such that $d(P) \geq 2\delta$. As pointed out in section 2, we have $\hat{h}(P) = 0$ if and only if $P \in \tilde{C}_K$. Thus

$$0 < h' \leq \min \{\hat{h}(P);\ P \in C_K \setminus \tilde{C}_K\}$$

as desired, the positivity of h' being a consequence of the facts that $\hat{h}$ is positive definite on $C_K \setminus \tilde{C}_K$ and that there are only finitely many points P in C_K satisfying $d(P) < 2\delta$ (see [8,14]).

For actually determining h', one has to approximately calculate $\hat{h}(P)$ for a finite number of points $P \in C_K \setminus \tilde{C}_K$. These points are found by trial and error as in the application of Manin's theorem. In order to approximately compute $\hat{h}(P)$ for a given point $P \in C_K \setminus \tilde{C}_K$, one can utilize the relation [14]

$$\hat{h}(P) = d(P) + \sum_{i=1}^{\infty} \frac{d(2^{i-1}P, 2^{i-1}P)}{2^{2i}}$$

in which the expressions

$$d(2^{i-1}P,\ 2^{i-1}P) = d(2^iP) - 4d(2^{i-1}P)$$

satisfy the inequalities (see appendix)

$$-15\left(\log 2 + \frac{\mu_\infty}{n}\right) - \frac{12}{n} \log B \leq d(2^{i-1}P, 2^{i-1}P) \leq 6\left(\log 2 + \frac{\mu_\infty}{4n}\right) + \frac{3}{n} \log B.$$

Hence, the degree of accuracy in the calculation of $\hat{h}(P)$ is evident from the estimate (see appendix)

$$-\frac{5}{2^{2m}}\left(\log 2 + \frac{\mu_\infty}{n}\right) - \frac{4}{n2^{2m}} \log B \leq \hat{h}(P) - d(P) - \sum_{i=1}^{m} \frac{d(2^{i-1}P, 2^{i-1}P)}{2^{2i}} \leq \frac{2}{2^{2m}}\left(\log 2 + \frac{\mu_\infty}{4n}\right) + \frac{1}{n2^{2m}} \log B\ .$$

An approximate calculation of $\hat{h}(P)$ for all $P \in C_K \setminus \tilde{C}_K$ such that $d(P) < 2\delta$ in accordance with the above explanations leads to the corresponding approximation of the searched value h', and this approximation can be carried through to a sufficient degree of accuracy.

5. Estimation of the determinant

In this section the most critical point of the attempted generalization of Manin's algorithm is reached. Estimating the determinant H of the elliptic curve C over K means a difficult problem because H is defined by virtue of a basis of the rational point group C_K (see section 1), whereas such a basis is just to be determined by the algorithm using H. Manin [7] solved this problem in the case $K = \mathbb{Q}$ under the condition that the Weil conjecture and the Birch and Swinnerton-Dyer conjecture are true. The determinant H is closely connected with the behavior of the L-series $L(C,s)$ of C over K near $s=1$ via the conjecture of Birch and Swinnerton-Dyer (see below). But for $K = \mathbb{Q}$ the function $L(C,s)$ and its derivatives $L^{(i)}(C,s)$ for $i = 1,\ldots,r'$ can be approximately evaluated at $s=1$ via the Weil conjecture. In this way the desired estimation of H is then accomplished.

Turning to the general case of an arbitrary finite-algebraic number field K over which the curve C is defined we still can utilize the relation, supplied by the conjecture of Birch and Swinnerton-Dyer, between the determinant H of C over K and the behavior of the L-series of C over K near $s=1$. But since an analogue of the Weil conjecture is lacking (see [11]), there seems to be no comparable tool for approximately evaluating $L(C,s)$ and $L^{(i)}(C,s)$ for $i = 1,\ldots,r'$ at $s=1$ in the general case.

Let us discuss the general situation in detail.

For each prime divisor $\mathfrak{p}$ of the number field K, we define the local L-series of the elliptic curve C over K as follows. Suppose first that C has good reduction modulo $\mathfrak{p}$ such that the reduced curve $C(\mathfrak{p})=C \bmod \mathfrak{p}$ is an elliptic curve defined over the residue field $K(\mathfrak{p})$ of K with respect to $\mathfrak{p}$. Recall that the norm $\mathfrak{N}\mathfrak{p}=p^{f_\mathfrak{p}}=q$ is just the cardinality of $K(\mathfrak{p})$ (see [4,12]), where $f_\mathfrak{p}=[K(\mathfrak{p}):\mathbb{F}_p]$ stands for the degree of $\mathfrak{p}$ over the prime p of $\mathbb{Q}$ such that $\mathfrak{p}|p$ ($\mathbb{F}_p$ being the prime field of characteristic p). The local L-series of the curve C over K at $\mathfrak{p}$ is now defined by

$$L_\mathfrak{p}(C,s) = \{(1-\alpha_\mathfrak{p} q^{-s})(1-\bar{\alpha}_\mathfrak{p} q^{-s})\}^{-1},$$

where the (complex) numbers $\alpha_\mathfrak{p}$ and $\bar{\alpha}_\mathfrak{p}$ are characterized by the relations

$$|\alpha_\mathfrak{p}|=|\bar{\alpha}_\mathfrak{p}|=\sqrt{q} \text{ and } N_\mathfrak{p}=1+q-\alpha_\mathfrak{p}-\bar{\alpha}_\mathfrak{p},$$

$N_\mathfrak{p}$ denoting the number of points of the reduced curve $C(\mathfrak{p})$ over the finite field $K(\mathfrak{p})$. Suppose next that C has bad reduction modulo $\mathfrak{p}$ such that $C(\mathfrak{p})=C \bmod \mathfrak{p}$ is a cubic defined over $K(\mathfrak{p})$ and having a singularity. Then we set

$$L_\mathfrak{p}(C,s)=(1-q^{-s})^{-1}, \ (1+q^{-s})^{-1}, \text{ or } 1,$$

according as the singularity of the reduced curve $C(\mathfrak{p})$ over $K(\mathfrak{p})$ is an ordinary double point with two (distinct) rational tangents, an ordinary double point with two irrational tangents, or a cusp, respectively. Again we call $L_\mathfrak{p}(C,s)$ the local L-series of the curve C over K at $\mathfrak{p}$.

Now the global L-series of the curve C over K is defined as the Euler product

$$L(C,s) = \prod_\mathfrak{p} L_\mathfrak{p}(C,s)$$

taken over all prime divisors (finite places) $\mathfrak{p}$ of K. The global L-series L(C,s) is known to converge in the complex half-plane Re $s>\frac{3}{2}$.

To state the conjecture of Birch and Swinnerton-Dyer, one has to presuppose that L(C,s) can be analytically continued over the whole complex plane or at least up to a neighbourhood of s=1. Furthermore, the Tate-Šafarevič group Ш of C over K must be assumed finite (see [7,1o, 11]).

Conjecture of Birch and Swinnerton-Dyer [7,1o,11]

The L-series of the elliptic curve C over K has the asymptotic expansion near s=1

$$L(C,s) \sim (s-1)^r \frac{|Ш|_H}{|\tilde{C}_K|^2} \cdot \frac{2^{r_2}}{\sqrt{|D|}} \cdot M,$$

where the bars signify cardinalities and M is the product of some local factors which correspond to a certain finite set of places of K including all the infinite places and those finite places $\mathfrak{p}$ at which C has bad reduction modulo $\mathfrak{p}$.

Taking the r-th derivatives on both sides, we derive from the Birch and Swinnerton-Dyer conjecture the asymptotic expansion near s=1

$$L^{(r)}(C,s) \sim r! \frac{|Ш|_H}{|\tilde{C}_K|^2} \cdot \frac{2^{r_2}}{\sqrt{|D|}} \cdot M.$$

Now suppose we know an upper bound Λ for the r-th derivative of L(C,s) at s=1 and a lower bound λ for the factor M, more precisely, suppose that

$$\left|\frac{1}{r!} L^{(r)}(C,1)\right| \le \Lambda \text{ and } M \ge \lambda > 0.$$

Then it follows from the asymptotic expansion of $L^{(r)}(C,s)$ near s=1 that we have

$$H \le 2^{-r_2} \lambda^{-1} \Lambda \, |\tilde{C}_K|^2 \sqrt{|D|}.$$

Thus we can choose this upper bound for H as the constant H' in Manin's theorem. In the case $K=\mathbb{Q}$ this choice for H' is exactly the same as the one made in [7,16].

However, as pointed out at the beginning of this section, the feasibility of

estimating $L^{(r)}(C,1)$ is an open problem in the general case of an elliptic curve C over a finite-algebraic number field $K \neq \mathbb{Q}$.

6. "Division by two" and the "infinite descent"

As soon as the constants in Manin's theorem are known, the points $P \in C_K \setminus \tilde{C}_K$ satisfying the inequality for d(P) in that theorem can be determined by the method indicated in section 1. Suppose we have found these points by trial and error. Denote them by $P_1,\ldots,P_m$. According to Manin's theorem, the points $P_1,\ldots,P_m$ generate a certain subgroup

$$\hat{C}_K^{(o)} \cong C_K^{(o)}/\tilde{C}_K$$

of finite index $\rho \leq r'!$ in a maximal free subgroup $\hat{C}_K \cong C_K/\tilde{C}_K$ of rank $r \leq r'$ in the rational point group C_K. The group $\hat{C}_K$ is to be determined in terms of a basis.

Next we factor out the subgroup $2\hat{C}_K$ of $\hat{C}_K$ consisting of all points $P \in \hat{C}_K$ such that $P=2Q$ for some $Q \in \hat{C}_K$. This yields

$$(\hat{C}_K^{(o)} + 2\hat{C}_K)/2\hat{C}_K \leq \hat{C}_K/2\hat{C}_K ,$$

where the factor group on the right has order 2^r. By a finite procedure of repeated "divisions by two" of the points in $\hat{C}_K^{(o)}$ we shall obtain a subgroup $\hat{C}_K^{(\varkappa)} \leq \hat{C}_K$ such that

$$\hat{C}_K^{(\varkappa)} + 2\hat{C}_K = \hat{C}_K \qquad (\varkappa \geq 0)$$

in the following manner.

We have $\hat{C}_K^{(o)} = \langle P_1,\ldots,P_m \rangle$. Apply the subsequent two steps executing the required divisions by two by means of the duplication formula for the point addition in C_K.

(1) Try to divide all points P_i by 2 $(i=1,\ldots,m)$. Suppose that $P_1,\ldots,P_\alpha$ are not divisible by 2, whereas $P_{\alpha+1},\ldots,P_m$ are divisible by 2. Then we have

$P_\nu = 2P_\nu'$ for some $P_\nu' \in C_K$ $(\nu=\alpha+1,\ldots,m)$.

(2) Try to divide all points P_i+P_j by 2 $(1 \leq i < j \leq \alpha)$.

Suppose that $P_1,\ldots,P_\beta$ are such that the P_i+P_j are not divisible by $2 (1 \leq i < j \leq \beta)$, whereas we have

$P_\nu = P_{j_\nu} + 2P_\nu'$ for some $P_\nu' \in C_K (\nu=\beta+1,\ldots,\alpha$; $1 \leq j_\nu \leq \beta)$.

Having carried out steps (1) and (2) we proceed as follows. For $\nu=1,\ldots,\beta$,we write $P_\nu'=P_\nu$. For $\nu=\beta+1,\ldots,\alpha,\ldots,m$, we replace P_ν by P_ν' whenever $P_\nu' \neq P_i'$ for $i=1,\ldots,\nu-1$, and we drop P_ν and P_ν' otherwise. After a suitable renumbering of the points P_ν' for $\nu=\beta+1,\ldots,\alpha,\ldots,m$, if need be, we obtain a subgroup $\hat{C}_K' = \langle P_1',\ldots,P_\beta',\ldots,P_{m'}' \rangle$ of index ρ' in $\hat{C}_K$ such that $\hat{C}_K^{(o)} \leq \hat{C}_K' \leq \hat{C}_K$, $\rho' | \rho$, and $m' \leq m$. Then we repeat steps (1) and (2) for the points $P_{\beta+1}',\ldots,P_{m'}'$ of the new group $\hat{C}_K'$ in place of $\hat{C}_K^{(o)}$, keeping in mind that the points $P_1',\ldots,P_\beta'$ have been treated already. Going on this way we end up, after a finite number of steps, with a certain subgroup $C_K^{(\varkappa)} = \langle P_1^{(\varkappa)},\ldots,P_{m^{(\varkappa)}}^{(\varkappa)} \rangle$ of $\hat{C}_K$ such that neither the $P_i^{(\varkappa)}$ $(i=1,\ldots,m^{(\varkappa)})$ nor the $P_i^{(\varkappa)}+P_j^{(\varkappa)}$ $(1 \leq i < j \leq m^{(\varkappa)})$ are divisible by 2, where $m^{(\varkappa)} \leq m$. Then, we conclude that $\hat{C}_K^{(\varkappa)}$ has odd index $\rho^{(\varkappa)}$ in $\hat{C}_K$, where $\rho^{(\varkappa)} | \rho$. Hence, we see that

$\hat{C}_K^{(\varkappa)} + 2\hat{C}_K = \hat{C}_K$ and, a fortiori,

$$(\hat{C}_K^{(\varkappa)} + 2\hat{C}_K)/2\hat{C}_K = \hat{C}_K/2\hat{C}_K$$

as desired. From the set of all points of the form $P_{i_1}^{(\varkappa)} + \ldots + P_{i_\nu}^{(\varkappa)} (1 \leq i_1 < \ldots < i_\nu \leq m^{(\varkappa)})$ in $\hat{C}_K^{(\varkappa)}$ we can therefore select a complete set of representatives in $\hat{C}_K$ of the 2^r cosets in the factor group $\hat{C}_K/2\hat{C}_K$. As a by-product of this process, we obtain the rank r of the curve C over K.

Let $\{Q_1,\ldots,Q_{2^r}\}$ be such a set of

representatives in $\hat{C}_K$ for $\hat{C}_K/2\hat{C}_K$. There is a point $Q_\nu (1\le\nu\le 2^r)$ such that

$$\hat{h}(Q_\nu) = \max_{i=1,\dots,2^r} \{\hat{h}(Q_i)\}.$$

We shall briefly write $Q=Q_\nu$. The "infinite descent" procedure now shows that the group $\hat{C}_K$ is generated by the set $\{Q_1,\dots,Q_{2^r}\}$ and the set $\{R_1,\dots,R_t\}$ of all points $R_j\in\hat{C}_K$ such that

$$\hat{h}(R_j) \le 1+\hat{h}(Q)$$

(see appendix), where the fact that $t<\infty$ was already noted in section 1. For actually finding all points $R_j\in\hat{C}_K$ satisfying this condition it suffices to determine the (possibly a little larger) finite set of all points $R_j\in\hat{C}_K$ such that

$$d(R_j) \le 1+2\log 2 + \delta + d(Q)$$
$$= 1+7\log 2+\frac{9}{2n}\mu_\infty+\frac{3}{n}\log B + d(Q).$$

This is clear from the estimate of the difference $d(P) - \hat{h}(P)$ given in section 4. The points $R_j\in\hat{C}_K$ fulfilling this latter condition are obtained by employing once more the trial-and-error method. In doing so, one can utilize earlier calculations in connection with Manin's theorem (section 1) and the theorem on the torsion subgroup $\tilde{C}_K$ (section 2).

Finally, we get a basis of the group $\hat{C}_K$ by applying the elementary divisor theorem to the set $\{Q_1,\dots,Q_{2^r},R_1,\dots R_t\}$ of generators of $\hat{C}_K$. Since the torsion subgroup $\tilde{C}_K$ of the rational point group C_K is known from section 2, one can exhibit a basis of the whole group $C_K=\tilde{C}_K\oplus\hat{C}_K$ itself.

7. Concluding remarks

The above exposition of a generalization of Manin's conditional algorithm is to be considered a modest first step towards the implementation of a procedure for determining the rank and a basis of the group C_K of rational points on an elliptic curve C over a finite-algebraic number field K. The amount of calculation involved will surely require the use of an electronic high speed computer. The algorithm is "conditional" insofar as its feasibility depends essentially on the truth of the conjecture of Birch and Swinnerton-Dyer which has to be taken for granted. The term "conditional" is moreover referring to the (open) problem of estimating the r-th derivative of the L-series L(C,s) of C over K at s=1. In the special case $K=\mathbb{Q}$, this problem can be solved by assuming the truth of the Weil conjecture as was shown by Manin [7]. The Weil conjecture guarantees in particular also that L(C,s) can be analytically continued over the whole complex plane, a property needed for stating the conjecture of Birch and Swinnerton-Dyer.

Next we remind of the condition, appearing in the theorem of section 3, according to which the l-division points of C over K have to belong to K for some prime number l. (But compare the remark at the end of section 3.)

Beyond the above-mentioned conditions, the feasibility of the generalized algorithm depends also on a rather good knowledge of the arithmetic of the number field K over which the elliptic curve C is defined.

For example, a complete system of integral divisors of K, representing the finitely many divisor classes of K, is used for writing the coordinates of the points $P\in C_K$ as quotients of integers in a suitable manner as required in the definition of d(P) (see section 1). Such a system can be obtained, e.g., by the method described in [13]. The same method yields also the class number and the structure of the class group $\mathfrak{C}(K)$ of the field K and, more generally, the S-class number and the structure of the S-class

group $\mathfrak{C}_S(K)$ of K for any finite set S of prime divisors of K. Then, the rank $rk_l\mathfrak{C}_S(K)$ of the l-Sylow subgroup of $\mathfrak{C}_S(K)$ can be computed for any prime number l. Of course, the quantity $rk_l\mathfrak{C}_S(K)$, appearing in the bound for the rank r of C over K in the theorem of section 3, does not exceed the l-rank $rk_l\mathfrak{C}(K)$ of the ordinary class group $\mathfrak{C}(K)$ (see [12]).

The definition of d(P) is furthermore based on the $n=r_1+2r_2$ distinct embeddings of the field K into the complex numbers $\mathbb{C}$ (see section 1). With these embeddings of K, we know the rank $rk\,u(K)=r_1+r_2$ of the unit group u(K) of K which also appears in the bound for the rank r of C over K given in the theorem of section 3.

For determining the torsion subgroup $\tilde{C}_K$ of C_K by virtue of the theorem of section 2, an explicit knowledge of the decomposition law [4,12], according to which the primes p of $\mathbb{Q}$ split in K, is helpful (compare [15]). The decomposition law is also useful as a tool in the calculation of the local L-series $L_p(C,s)$ of C over K. Specifically, the discriminant D of the field K must be computed. We have seen that D occurs in the Minkowski bound of K and thus in the bound measuring the deviation of the modified Weil height d from the Néron-Tate height $\hat{h}$ on C_K.

In passing, we direct attention to the remarkable fact that Manin's conditional algorithm is the first procedure devised for determining a basis of the rational point group C_K of the curve C over the number field K.

A p p e n d i x

We shall prove here those assertions which were left without proof in the above exposition, thus verifying also the corresponding unproved assertions in [16].

To section 1

Let us begin with establishing the representation required in section 1 for the coordinates of the rational points $P=(\xi,\eta)$ in C_K such that $\xi\neq 0$, $\eta\neq 0$. We follow closely [5].

The Weierstrass equation of C applied to such a point P in C_K yields divisor representations of its coordinates in K of the form

$$\xi \cong \frac{\mathfrak{x}}{\mathfrak{z}^2} \quad \text{and} \quad \eta \cong \frac{\mathfrak{y}}{\mathfrak{z}^3}$$

with some integral divisors $\mathfrak{x},\mathfrak{y}$, and $\mathfrak{z}$ of K having the g.c.d.'s

$$(\mathfrak{x},\mathfrak{z}) = (\mathfrak{y},\mathfrak{z}) = 1.$$

Choose a fixed system V of representatives for the (finitely many) divisor classes of K such that V is made up of integral divisors c of K whose norms do not exceed the Minkowski bound B of K (see [4,12,13]),

$$\mathfrak{N}c \leq B$$

Then, for the given divisor $\mathfrak{z}$ of K, there exists a divisor $c\in V$ such that $\mathfrak{z}c$ is a principal divisor of K,

$$\mathfrak{z}c \cong z \quad \text{for some nonzero } z\in K.$$

Hence we obtain the representations

$$\xi \cong \frac{\mathfrak{x}c^2}{\mathfrak{z}^2c^2} \cong \frac{x'}{z^2} \quad \text{and} \quad \eta \cong \frac{\mathfrak{y}c^3}{\mathfrak{z}^3c^3} \cong \frac{y'}{z^3} \quad \text{for some}$$

nonzero $x',y' \in K$.

On multiplying x' and y' by suitably chosen units of K, we end up with two nonzero elements x and y in K which furnish the desired representations

$$\xi = \frac{x}{z^2} \quad \text{and} \quad \eta = \frac{y}{z^3}$$

of the coordinates of $P\in C_K$ as quotients of integers x, z^2 and y, z^3 in K having the g.c.d.'s

$$(x,z^2) = c^2 \quad \text{and} \quad (y,z^3) = c^3.$$

To section 4

(a) We wish to derive the estimate claimed for the difference of the modified Weil height d and the Néron-Tate height $\hat{h}$ on the group C_K. In what follows we shall use the notation $p|\infty$ or $p\nmid\infty$ for the infinite or the finite places p of K, writing in the latter case also $p|p$ instead of $p\nmid\infty$ in order to signify the prime p of $\mathbb{Q}$ over which p lies. Let v_p, normalized as in [14], designate the absolute value on K corresponding to the (finite or infinite) place p of K. Observe that this normalization of v_p differs from the normalization of the valuation w_p used in section 2 for the finite places p of K. In fact, we have, for each nonzero $c\in K$, the relation

$$v_p(c)=\frac{1}{e_p}w_p(c)\log p \text{ whenever } p|p$$

(see [4]), where e_p denotes the ramification index of p. As in [14] we now introduce the real numbers

$$\mu_p=\min\{\tfrac{1}{2}v_p(a),\ \tfrac{1}{3}v_p(b)\} \text{ for each place } p \text{ of K.}$$

Notice that the numbers μ_p differ, for each place p of K such that $p|p$, only by a factor $\frac{1}{2e_p}\log p$ or $\frac{1}{3e_p}\log p$ from the numbers $\bar{\mu}_p$ introduced in section 2. For each place p of K, let $n_p=[K_p:\mathbb{Q}_p]$ or $=[K_p:\mathbb{R}]$ denote the local degree of p according as $p|p$ or $p|\infty$ respectively (see [4,12,14]). Then we put as in [14]

$$\mu=-\sum_p n_p\mu_p\ ,$$

the sum being taken over all (finite and infinite) places p of K.

The function used in [14] for defining the Néron-Tate height $\hat{h}$ on C_K was

$$d_1(P)=\begin{cases}-\frac{3}{2n}\Sigma n_p\min\{\mu_p,v_p(\xi)\} & \text{if } P=(\xi,\eta)\neq 0\\ \frac{3}{2n}\mu & \text{if } P=0\end{cases},$$

where $n=[K:\mathbb{Q}]$ is the field degree of K over $\mathbb{Q}$ as before. (Actually, we used in [14] the function nd_1 and called it d). The difference function $d_1-\hat{h}$ on C_K was shown in [14] to fulfill the estimate

$$-2\log 2\le d_1(P)-\hat{h}(P)\le 5\log 2+\frac{3}{n}\mu \text{ for } P\in C_K.$$

Starting off with this estimate we are left with the task of estimating the difference function $d-d_1$ on C_K. The modified Weil height d on C_K was introduced in section 1 as

$$d(P)=\frac{3}{2n}\log\prod_{\nu=1}^{n}\max\{\sqrt[2]{|a^{(\nu)}|}\,|z^{(\nu)}|^2\ ,\ \sqrt[3]{|b^{(\nu)}|}\,|z^{(\nu)}|^2\ ,\ |x^{(\nu)}|\}$$

with the coordinates of the point $P=(\xi,\eta)$ in C_K written in the shape

$$\xi=\frac{x}{z^2} \text{ and } \eta=\frac{y}{z^3} \text{ (x,y,and z integers in K)}$$

as described in section 1 and at the beginning of this appendix. Hence, we obtain in the notation of [14] (making the convention that $\log 0=-\infty$)

$$\begin{aligned}d(P)&=\frac{3}{2n}\sum_{\nu=1}^{n}\max\{\tfrac{1}{2}\log|a^{(\nu)}|+2\log|z^{(\nu)}|,\\ &\tfrac{1}{3}\log|b^{(\nu)}|+2\log|z^{(\nu)}|,\log|x^{(\nu)}|\}\\ &=\frac{3}{2n}\sum_{p|\infty}n_p\max\{\tfrac{1}{2}\log|a|_p+2\log|z|_p\ ,\\ &\tfrac{1}{3}\log|b|_p+2\log|z|_p,\ \log|x|_p\}\\ &=-\frac{3}{2n}\sum_{p|\infty}n_p\min\{\tfrac{1}{2}v_p(a)+2v_p(z),\tfrac{1}{3}v_p(b)+\\ &+2v_p(z),\ v_p(x)\}\\ &=-\frac{3}{2n}\sum_{p|\infty}n_p\min\{\mu_p+2v_p(z),\ v_p(x)\}.\end{aligned}$$

On the other hand, the sum formula for the absolute values v_p on K (see [14]) yields for the function d_1 on C_K

$$\begin{aligned}d_1(P)&=-\frac{3}{2n}\Sigma n_p\min\{\mu_p,v_p(\tfrac{x}{z^2})\}\\ &=-\frac{3}{2n}\Sigma n_p\min\{\mu_p+2v_p(z),\ v_p(x)\}.\end{aligned}$$

A comparison of the last expressions for the functions d and d_1 on C_K shows at once that for $P \in C_K$

$$d_1(P) = d(P) - \frac{3}{2n} \sum_{p \nmid \infty} n_p \min\{\mu_p + 2v_p(z), v_p(x)\}.$$

Hence, taking into account that a,b,x, and z are integers in K, we obtain the inequality

$$d_1(P) \leq d(P) \text{ for } P \in C_K.$$

It remains to establish an inequality in the opposite direction between $d_1(P)$ and d(P) up to an additive constant not depending on the point P. We shall explicitly exhibit such a constant. Remembering the fact that $\mu_p \geq 0$ for each finite place p of K and keeping in mind the choice of the coordinate $\xi = \frac{x}{z}$ of P such that $(x,z^2) = c^2$ with $c \in V$, we derive from the above equation between $d_1(P)$ and d(P) the inequality

$$d_1(P) \geq d(P) - \frac{3}{2n} \sum_{p \nmid \infty} n_p \min\{\mu_p + 2v_p(z), \mu_p + v_p(x)\}$$

$$= d(P) - \frac{3}{2n} \sum_{p \nmid \infty} n_p \mu_p - \frac{3}{2n} \sum_{p \nmid \infty} n_p \min\{2v_p(z), v_p(x)\}$$

$$= d(P) - \frac{3}{2n} \sum_{p \nmid \infty} n_p \mu_p - \frac{3}{n} \sum_{p \nmid \infty} n_p v_p(c).$$

Consider the second and the third term of the last expression.

As regards the second term, we have in the notation of [14] and section 4

$$0 \geq -\frac{3}{2n} \sum_{p \nmid \infty} n_p \mu_p = \frac{3}{2n} \mu + \frac{3}{2n} \sum_{p \mid \infty} n_p \mu_p$$

$$= \frac{3}{2n} \mu + \frac{3}{2n} \sum_{p \mid \infty} n_p \min\{\tfrac{1}{2} v_p(a), \tfrac{1}{3} v_p(b)\}$$

$$= \frac{3}{2n} \mu - \frac{3}{2n} \sum_{p \mid \infty} n_p \max\{\tfrac{1}{2} \log|a|_p, \tfrac{1}{3} \log|b|_p\}$$

$$= \frac{3}{2n} \mu - \frac{3}{2n} \sum_{\nu=1}^{n} \max\{\tfrac{1}{2} \log|a^{(\nu)}|, \tfrac{1}{3} \log|b^{(\nu)}|\}$$

$$= \frac{3}{2n} \mu - \frac{3}{2n} \log \prod_{\nu=1}^{n} \max\{\sqrt[2]{|a^{(\nu)}|}, \sqrt[3]{|b^{(\nu)}|}\}$$

$$= \frac{3}{2n} \mu - \frac{3}{2n} \mu_\infty.$$

In particular, dropping the factor $\frac{3}{2n}$ and applying the sum formula for the absolute values v_p on K, we obtain

$$\mu_\infty \geq \mu = -\sum_p n_p \mu_p = -\sum_p n_p \min\{\tfrac{1}{2} v_p(a), \tfrac{1}{3} v_p(b)\}$$

$$\geq \begin{cases} -\frac{1}{2} \sum_p n_p v_p(a) & \text{if } a \neq 0 \\ -\frac{1}{3} \sum_p n_p v_p(b) & \text{if } b \neq 0 \end{cases} = 0.$$

Hence, the above second term under consideration satisfies the inequality

$$-\frac{3}{2n} \sum_{p \nmid \infty} n_p \mu_p \geq -\frac{3}{2n} \mu_\infty.$$

Now let us look at the third term $-\frac{3}{n} \sum_{p \nmid \infty} n_p v_p(c)$ of the above expression.

Since $c \in V$, the norm of c does not exceed the Minkowski bound B,

$$\mathfrak{N}c \leq B.$$

For the time being we shall denote, for each finite place p of K, by p_p the prime of $\mathbb{Q}$ such that $p \mid p_p$. Then, the norm inequality can be explicitly written as (see [4,12])

$$\mathfrak{N}c = \prod_{p \nmid \infty} p_p^{f_p w_p(c)} \leq B,$$

where f_p as usual stands for the degree of p over p_p (see section 5).

Taking logarithms on both sides, we get

$$\log \mathfrak{N}c = \sum_{p \nmid \infty} f_p w_p(c) \log p_p \leq \log B.$$

Hence, since $n_p = e_p f_p$ with the ramification index e_p of p, we have

$$\sum_{p \nmid \infty} n_p v_p(c) = \sum_{p \nmid \infty} \frac{1}{e_p} n_p w_p(c) \log p_p =$$

$$= \sum_{p \nmid \infty} f_p w_p(c) \log p_p \leq \log B.$$

The above third term under consideration thus satisfies the inequality

$$-\frac{3}{n} \sum_{p \nmid \infty} n_p v_p(c) \geq -\frac{3}{n} \log B.$$

In sum, we have established the estimate

$$d_1(P)\geq d(P)-\frac{3}{2n}\mu_\infty-\frac{3}{n}\log B \text{ for } P\in C_K.$$

Combining this estimate with the inequality $d_1(P)\leq d(P)$ derived before we obtain the desired estimate

$$0\leq d(P)-d_1(P)\leq \frac{3}{2n}\mu_\infty+\frac{3}{n}\log B \text{ for } P\in C_K.$$

In combination with the above-quoted estimate for the difference $d_1-\hat{h}$ on C_K, this result leads to the asserted inequalities

$$-2\log 2\leq d(P)-\hat{h}(P)\leq 5\log 2+\frac{3}{n}\mu+\frac{3}{2n}\mu_\infty$$

$$+\frac{3}{n}\log B$$

$$\leq 5\log 2+\frac{9}{2n}\mu_\infty+\frac{3}{n}\log B \text{ for } P\in C_K,$$

where in the last step we have again employed the relation $\mu\leq\mu_\infty$ which was shown above to be valid.

(b) Now we shall prove the estimate indicated in section 4 for the expression $d(P,P)=d(2P)-4d(P)$ as P ranges over the group C_K. To begin with we quote from [14] for the analog expression $d_1(P,P)=d_1(2P)-4d_1(P)$ the estimate

$$-(15\log 2+\frac{9}{n}\mu)\leq d_1(P,P)\leq 6\log 2 \text{ for } P\in C_K.$$

(Actually, we estimated in [14] the expression $nd_1(P,P)$ and denoted it by $d(P,P)$). On the other hand, it follows by definition of $d(P,P)$ and $d_1(P,P)$ from the estimate established in part (a) for the difference function $d-d_1$ on C_K, that we have

$$d_1(P,P)-\frac{6}{n}\mu_\infty-\frac{12}{n}\log B\leq d(P,P)\leq d_1(P,P)+$$

$$\frac{3}{2n}\mu_\infty+\frac{3}{n}\log B \text{ for } P\in C_K.$$

On replacing $d_1(P,P)$ by its cited lower and upper bound from [14], respectively, and remembering that $\mu\leq\mu_\infty$, we get

$$-15(\log 2+\frac{\mu_\infty}{n})-\frac{12}{n}\log B\leq d(P,P)\leq$$

$$\leq 6(\log 2+\frac{\mu_\infty}{4n})+\frac{3}{n}\log B \text{ for } P\in C_K$$

as asserted in section 4.
In particular, this yields for the sum

$$\hat{h}(P)-d(P)-\sum_{i=1}^{m}\frac{d(2^{i-1}P,2^{i-1}P)}{2^{2i}}$$

$$=\sum_{i=m+1}^{\infty}\frac{d(2^{i-1}P,2^{i-1}P)}{2^{2i}}=\frac{1}{2^{2m}}\sum_{i=1}^{\infty}$$

$$\frac{d(2^{m+i-1}P,2^{m+i-1}P)}{2^{2i}}$$

the asserted estimate (compare [14])

$$-\frac{5}{2^{2m}}(\log 2+\frac{\mu_\infty}{n})-\frac{4}{n2^{2m}}\log B\leq\frac{1}{2^{2m}}\sum_{i=1}^{\infty}$$

$$\frac{d(2^{m+i-1}P,2^{m+i-1}P)}{2^{2i}}\leq\frac{2}{2^{2m}}(\log 2+\frac{\mu_\infty}{4n})$$

$$+\frac{1}{n2^{2m}}\log B \text{ for } P\in C_K.$$

<u>To section 6</u> Let us show how to get a set of generators for the group $\hat{C}_K$ by applying the method of "infinite descent" (see [8]) to a given set $\{Q_1,\ldots,Q_{2^{r2^r}}\}$ of representatives in $\hat{C}_K$ of the 2^{r2^r} cosets in the factor group $\hat{C}_K/2\hat{C}_K$.

We have

$$\hat{C}_K/2\hat{C}_K=\{Q_1,\ldots,Q_{2^{r2^r}}\}/2\hat{C}_K .$$

The "infinite descent" works as follows. Let $R\in\hat{C}_K$ be an arbitrary rational point. Put $R_0=R$. There is a rational point $R_1\in\hat{C}_K$ such that

$$R_0=Q_{\nu_0}+2R_1 \text{ for some index } \nu_0 \text{ in } 1\leq\nu_0\leq 2^{r2^r}.$$

Furthermore, there is a rational point $R_2\in\hat{C}_K$ such that

$$R_1=Q_{\nu_1}+2R_2 \text{ for some index } \nu_1 \text{ in } 1\leq\nu_1\leq 2^{r2^r}.$$

Going on this way we obtain an infinite sequence $R_0,R_1,\ldots,R_{j-1},R_j,\ldots$ of rational points in $\hat{C}_K$ sucht that

$R_{j-1}=Q_{\nu_{j-1}}+2R_j$ for some index ν_{j-1} in $1\le\nu_{j-1}\le 2^r$, where $j\ge 1$.

Fortunately, only a finite portion of this infinite sequence is needed for finding a basis of $\hat{C}_K$ as we shall see. By induction on j, it follows from the last relation that the given rational point $R\in\hat{C}_K$ admits a representation of the form

$$(*)\quad R=2^jR_j+2^{j-1}Q_{\nu_{j-1}}+\ldots+2Q_{\nu_1}+Q_{\nu_o}\quad (j\ge 1).$$

We wish to estimate the value of the Néron-Tate height $\hat{h}$ at the point R_j. To this end we utilize the property of $\hat{h}$ to be a positive definite quadratic form on the group $\hat{C}_K$ (see [8,14]). Applying the function $\hat{h}$ to the relation $2R_j=R_{j-1}-Q_{\nu_{j-1}}$ we obtain for $j\ge 1$

$$4\hat{h}(R_j)=2\hat{h}(R_{j-1})+2\hat{h}(Q_{\nu_{j-1}})-\hat{h}(R_{j-1}+Q_{\nu_{j-1}})$$
$$\le 2\hat{h}(R_{j-1})+2\hat{h}(Q_{\nu_{j-1}}).$$

In this inequality we may replace $Q_{\nu_{j-1}}$ by the point Q chosen in section 6 in order to gain

$$4\hat{h}(R_j)\le 2\hat{h}(R_{j-1})+2\hat{h}(Q)\quad (j\ge 1).$$

Now we use induction on j to infer from this the new inequality

$$\hat{h}(R_j)\le\hat{h}(Q)+\frac{1}{2^j}\{\hat{h}(R)-\hat{h}(Q)\}\quad (j\ge 1).$$

When we choose the index j sufficiently large, we arrive at a rational point $R_j\in\hat{C}_K$ such that

$$\hat{h}(R_j)\le 1+\hat{h}(Q)=1+\max_{i=1,\ldots,2^r}\{\hat{h}(Q_i)\}.$$

Thus the given point $R\in\hat{C}_K$ can be represented as a linear combination (*) of the points $Q_1,\ldots,Q_{2^r}$ and a point R_j satisfying the above inequality for its $\hat{h}$-value. By virtue of the estimate for the difference function $d-\hat{h}$ on $\hat{C}_K$, this last inequality turns into the corresponding inequality with the Néron-Tate height $\hat{h}$ replaced by the modified Weil height d.

References

1. V.A. Dem'janenko, Torsion of elliptic curves. Izv. Akad. Nauk SSSR, Ser. Mat. 35 (1971), 28o-3o7 = Math. USSR Izvestija 5 (1971), 289-318.
2. G. Frey, Some remarks concerning points of finite order on elliptic curves over global fields. To appear.
3. H. Hasse, Simultane Approximation algebraischer Zahlen durch algebraische Zahlen. Monatsh. Math. Phys. 48 (1939), 2o5-225.
4. H. Hasse, Zahlentheorie. Akademie-Verlag, Berlin 1969.
5. J. Heuss, Zum schwachen Endlichkeitssatz von Mordell. Diploma Thesis, Karlsruhe 1975.
6. T. Honda, Isogenies, rational points and section points of group varieties. Japanese J.Math. 3o (196o), 84-1o1.
7. Ju.I. Manin, Cyclotomic fields and modular curves. Uspehi Mat. Nauk 26 (1971) no. 6 (162), 7-71 = Russian Math. Surveys 26 (1971) no. 6, 7-78.
8. D. Mumford, Abelian Varieties. Oxford University Press, London 1974.
9. J.-P. Serre, p-Torsion des courbes elliptiques (d'après Ju.I. Manin). In: Sém. Bourbaki, 22e année, 1969/7o, no. 38o, 281-294.
1o. H.P.F.Swinnerton-Dyer, The conjectures of Birch and Swinnerton-Dyer, and of Tate. In: Proc. Conf. on Local Fields, 132-157, Springer-Verlag, Berlin-Göttingen-Heidelberg 1967.
11. H.P.F.Swinnerton-Dyer and B.J.Birch, Elliptic curves and modular functions.

In: Modular Functions of One Variable IV, Lecture Notes in Math., vol. 476, pp. 2-32, Springer-Verlag, Berlin-Heidelberg-New York 1975.

12. E. Weiss, Algebraic Number Theory, McGraw-Hill Book Company, New York 1963.

13. H.G. Zimmer, Some computational aspects of, and the use of computers in, algebraic number theory. Computing 8 (1971), 363-381.

14. H.G. Zimmer, On the difference of the Weil height and the Néron-Tate height. Math. Z. 147 (1976), 35-51.

15. H.G. Zimmer, Points of finite order on elliptic curves over number fields. To appear in Arch. d. Math.

16. H.G. Zimmer, On Manin's conditional algorithm. To appear in Proc.Conf. on "Utilisation des calculateurs en mathématiques pures", Limoges 1975.

Computers in the Study of Lie Algebras

Bernard Kolman
Department of Mathematics
Drexel University

Robert E. Beck
Department of Mathematics
Villanova University

1. Introduction

Computers have been successfully used in group theory for a good many years to calculate the structure of a group. They have also been used to either prove or disprove a number of conjectures since they allow examination of large examples. Moreover, computers have been instrumental in the discovery of a number of new groups.

Few areas in algebra have been applied as extensively to other areas of mathematics and the physical sciences as Lie algebras. Applications almost invariably call for extensive computational work. There are strong parallels between the structure of groups and the structure theory of Lie algebras. In view of the successful application of computers to handle computational problems in group theory, it was reasonable to consider an analogous approach in Lie algebras.

Some of the difficulties encountered in doing computational work in Lie algebras are: (a) a Lie algebra is a vector space which has infinitely many elements if the field is infinite, (b) small perturbations in the multiplication table produce drastic changes in the structure of the Lie algebra, and (c) most of Lie algebra theory is existential. Thus, there are some limits on the questions that can be studied in Lie algebras by using a computer. Also, at times such a study must use a somewhat different point of view from that used in group theory.

In this paper we present a survey of the work to date in using computers to study Lie algebras. Computational problems in Lie algebras can be divided into three categories:

I. Representation Theory

II. Internal Structures
   A. Questions about the algebra itself
   B. Substructures

III. External Structures

In the second section of this paper we present a quick survey of the basic facts of Lie algebras. For further details, the reader is referred to [4,7]. The earliest use of computers in Lie algebras took place in representation theory. The impetus here was provided by

the pressing need for computational results in theoretical physics. Accordingly, in Section 3 we discuss the use of computers to handle the three basic problems in representation theory: the calculation of the inner multiplicities of a representation of a complex simple Lie algebra, the calculation of the outer multiplicities of the tensor product of two irreducible representations of a complex simple Lie algebra, and the calculation of the branching multiplicities of the restriction of an irreducible representation of a complex simple Lie algebra to a semisimple subalgebra. We briefly describe the computer implementations of the formulas of Kostant, Freudenthal and Racah for calculating inner multiplicities and compare these from a computational point of view. We also compare the implementations of the formulas of Steinberg and Racah for calculating the outer multiplicities. Finally, we describe a computer implementation for calculating branching multiplicities.

The fourth section describes more recent work being carried out in category II. The objective here is to provide a structural analysis of a given Lie algebra and to determine, for example whether L is nilpotent, solvable, semisimple, etc.

Section 5 deals with category III; here we discuss computer approaches to the calculation of cohomology groups, derivation algebras, and invariants of a Lie algebra.

2. A Survey of Lie Algebras

A. Structure Theory

A Lie algebra L is a finite-dimensional vector space over a field F in which there is defined a multiplication operation that associates with each x and y in L an element in L denoted by [x,y]. This operation satisfies the following properties:

$[x,y+z] = [x,y] + [x,z]$, $[x,cy] = c[x,y]$ where c is in F; (1)

$[x,x] = 0$ (2)

$[[x,y],z] + [[y,z],x] + [[z,x],y] = 0$ (3)

Properties (1) and (2) state that the product rule in L is bilinear and antisymmetric. Property (3) is known as the Jacobi identity. A familiar example of a Lie algebra is provided by real three-dimensional space with the usual vector cross product. This Lie algebra is denoted by so(3,R). Another important example is the set of all $n \times n$ matrices with the Lie product defined by $[A,B] = AB - BA$, where AB is the ordinary matrix product of A and B. If $\{e_1, e_2, \cdots, e_n\}$ is a basis for L, then properties (1) and (2) allow the multiplication in L to be completely specified by the products $[e_i, e_j], i<j$. We write $[e_i, e_j] = \sum_k c_{ij}^k e_k$. The c_{ij}^k are called the structure constants of L. A subalgebra S of the Lie algebra L is a subspace which is closed under the bracket operation of L; that is, if x and y are in S, then $[x,y]$ is in S. An ideal S of a Lie algebra L is a subalgebra which satisfies the additional property that if $x \in S$ and $y \in L$, then $[x,y] \in S$. The F-vector space spanned by $\{x_1, x_2, \cdots, x_r\}$ will be denoted by $\langle x_1, x_2, \cdots, x_r \rangle$. If U and V are subspaces of L, then the subspace of L spanned by all $[u,v], u \in V, v \in V$ is denoted by $[U,V]$.

Certain sequences of ideals play important roles in determining the structure of L. Let $L^{(0)} = L$, $L^{(1)} = [L,L], \cdots, L^{(k+1)} = [L^{(k)}, L^{(k)}]$. Each $L^{(i)}$ is an ideal in L and $L^{(i)} \supset L^{(i+1)}$. This sequence of ideals is called the derived series of L. Let $L^0 = L, L^1 = [L,L], \cdots, L^{k+1} = [L^k, L]$. Each L^i is an ideal of L and $L^i \supset L^{i+1}$. This sequence of ideals is called the lower central series of L. If for some $k, L^{(k)} = 0$, then L is called solvable. If for some $k, L^k = 0$, then L is called nilpotent.

Clearly $L^{(i)} \subset L^i$, which implies that every nilpotent Lie algebra is solvable. The converse is false.

Certain ideals of a Lie algebra are singled out for special attention. The center of L, Z, is defined as $Z = \{x \in L \mid [x,y] = 0 \text{ for all } y \in L\}$. The Lie algebra L is said to be abelian if $Z = L$; that is, if $[x,y] = 0$ for all $x,y \in L$. It can be shown that L has a unique maximal solvable ideal, R, called the radical of L, and a unique maximal nilpotent ideal, N, called the nil radical of L.

A simple Lie algebra is a Lie algebra L whose only ideals are the trivial ones: 0 and L. A Lie algebra is called semisimple if it is a direct sum of simple ideals; thus, the study of semisimple Lie algebras reduces to the study of simple Lie algebras. Of course, if L is simple it is also semisimple. There are four main classes of simple Lie algebras over the complex numbers: $A_n (n \geq 1)$, $B_n (n \geq 2)$, $C_n (n \geq 3)$ and $D_n (n \geq 4)$, and five exceptional ones: F_2, G_4, E_6, E_7, and E_8. These Lie algebras can be completely determined by their corresponding Coxeter-Dynkin diagrams.

A derivation of L is a linear transformation $D: L \to L$ such that $D[x,y] = [Dx,y] + [x,Dy]$. For any $x \in L$, the mapping ad $x: L \to L$ defined by $(\text{ad } x)(y) = [x,y]$ for y in L, is a derivation, called the adjoint mapping. For some Lie algebras there are derivations which are not of the form ad x. The Killing form is a bilinear form defined on L by

$$(x,y) = \text{Tr}(\text{ad } x \circ \text{ad } y)$$

where Tr denotes the trace of the composition of the linear transformations ad x and ad y for x and y in L. If the characteristic of F is zero, then the Lie algebra L is semisimple if and only if its Killing form is nondegenerate and the radical of L is the orthogonal complement of [L,L] with respect to the Killing form.

If K is a subalgebra of L, the normalizer $N_L(K)$ is defined as $N_L(K) = \{x \in L \mid [x,L] \subset K\}$. It is the largest subalgebra of L which includes K as an ideal. For example, if K is an ideal, then $N_L(K) = L$. A Cartan subalgebra H of L is a subalgebra of L which is nilpotent and equals its normalizer in L. Cartan subalgebras are not unique but they all have the same dimension ℓ, called the rank of L. If H is a Cartan subalgebra of a complex semisimple Lie algebra L, a root α is a nonzero linear form on H with the property that there exists a nonzero vector e_α in L such that $[h,e_\alpha] = \alpha(h)e_\alpha$ for all h in H. Thus $\alpha(h)$ is an eigenvalue of the linear transformation ad h. It can be shown that the root space of α, consisting of all the vectors e_α satisfying $(\text{ad } h)e_\alpha = \alpha(h)e_\alpha$ for all h in H, is a one-dimensional subspace of L.

Since the Killing form of a complex semisimple Lie algebra L is nondegenerate on any Cartan subalgebra H, it can be used to identify H with its dual H* by letting α in H* correspond to the unique vector h_α in H such that $\alpha(h) = (h_\alpha, h)$ for all h in H. Having made this identification, a bilinear form can be defined on H* by letting

$$(\alpha,\beta) = (h_\alpha, h_\beta) \text{ for all } \alpha,\beta \text{ in } H^*.$$

Let H_R^* be the ℓ-dimensional real vector space formed by restricting the scalar field of H* to the real numbers. Then H_R^* consists of all linear combinations of roots of L with real coefficients. The bilinear form on H* restricted to H_R^* is real and positive-definite, thus introducing an inner product into H_R^*. Furthermore, there is a set of ℓ roots, called simple roots, which is a basis for H_R^* such that any root α can be written as $\alpha = \sum_{i=1}^{\ell} k_i \alpha_i$, where the k_i are either all nonnegative or all nonpositive integers and $\{\alpha_i\}$ is the set of simple roots.

Example 1. Consider the 6-dimensional Lie algebra whose multiplication is defined by

$$[e_1,e_5]=e_3,\ [e_2,e_6]=e_1,\ [e_3,e_4]=e_2$$
$$[e_4,e_2]=e_3,\ [e_4,e_5]=e_6,\ [e_5,e_3]=e_1$$
$$[e_5,e_6]=e_4,\ [e_6,e_1]=e_2,\ [e_6,e_4]=e_5.$$

We see that $L^{(1)} = L^1 = [L,L] = L$, so that L is neither solvable nor nilpotent. The subspace spanned by $\{e_1,e_2,e_3\}$ is an ideal so that L is not simple. We find

$$Z = 0$$
$$R = \langle e_1,e_2,e_3 \rangle$$
$$N = \langle e_1,e_2,e_3 \rangle$$

and a Cartan subalgebra is $\langle e_1,e_4 \rangle$. The only pairs of basis elements for which the Killing form is not zero are $(e_4,e_4) = (e_5,e_5) = (e_6,e_6) = -4$.

B. Representation Theory

A representation of a Lie algebra L is a pair (ρ,V), where V is a finite-dimensional vector space and $\rho: L \to gl(V)$ is a homomorphism of L into the Lie algebra gl(V) of linear transformations on V. The vector space V is called the representation space. The Lie product in gl(V) is defined by $[A,B] = AB - BA$ where AB means the composition of the linear transformations A and B. Since $[ad(a),ad(b)] = ad([a,b])$, (ad,L) is a representation called the adjoint representation of L.

Now let (ρ,V) be a representation of L. A subspace W of V is called invariant if $\rho(L)W \subset W$. We say that (ρ,V) is irreducible if the only invariant subspaces of V are 0 and V itself. The representation is called completely reducible if V is a direct sum of irreducible invariant subspaces. If L is semisimple, then every (finite-dimensional) representation of L is completely reducible. This means that the irreducible representations of L are the basic building blocks for the representation of L.

For the rest of this section we shall restrict our remarks to irreducible representations of complex simple Lie algebras. If H* is the dual of the Cartan subalgebra H of L, then a linear form $\mu \in H^*$ is called a weight of (ρ,V) if there exists a nonzero vector v in V such that $\rho(h)v = \mu(h)v$ for all $h \in H$. The vector v is called a weight vector associated with the weight μ. Thus, v is simultaneous eigenvector for all linear transformations $\rho(h)$, and $\mu(h)$ is the corresponding eigenvalue. The set of all weight vectors together with zero is an invariant subspace of V denoted by V_μ and called the weight space associated with the weight μ. The representation ρ has a finite number of weights. The vector space V has a basis of weight vectors and

$$V = V_{\mu_1} \oplus V_{\mu_2} \oplus \cdots \oplus V_{\mu_k}.$$

Moreover, $\mu(h)$ is the only eigenvalue of the linear transformation $\rho(h)$ when restricted to V_μ. If ρ is the adjoint representation of L, then a weight is merely a root α and its weight space is the root space L_α. The Cartan subalgebra H itself is just the weight space corresponding to the zero weight.

The set of all weights associated with a given representation ρ of L is called the weight system of ρ. The dimension n_μ of V_μ is called the inner multiplicity of μ. If μ is not a weight, we define its inner multiplicity n_μ to be 0. The character χ of ρ is defined by

$$\chi(h) = \operatorname{tr}(\exp \rho(h)) = \sum_{\mu \in \Lambda} n_\mu e^{\mu(h)}, h \in H$$

where Λ is the weight system associated with ρ. Both weights and roots are vectors in H*, the dual space of the Cartan subalgebra H. Since the weights are real linear combinations of the roots, the weight system lies in H_R^*. The dimension of H_R^* is ℓ, the rank of L. H_R^* can be ordered lexicographically by

agreeing that ξ is positive if the first nonzero component of ξ, when it is written in terms of the simple roots, is positive and we write $\xi > 0$. Thus, the weight system of ρ is ordered, and since it is finite there is a weight which is higher than all the others. This weight λ is called the highest weight of the representation, and its multiplicity is one. An irreducible representation of a semisimple Lie algebra is completely determined by its highest weight.

The set S of vectors in H_R^* which can be highest weights for irreducible representations is closed under addition. A highest weight is called fundamental if it is not the sum of two other highest weights. If L is a semisimple Lie algebra of rank ℓ, then S contains a basis for H_R^* consisting of ℓ fundamental weights $\lambda_1,\cdots,\lambda_\ell$. These fundamental weights can be indexed so that if $\alpha_1,\cdots,\alpha_\ell$ is a set of simple roots then

$$2(\lambda_i,\alpha_j)/(\alpha_i,\alpha_j) = \delta_{ij}$$

where δ_{ij} is the Kronecker delta. Thus, every weight μ can be written as a linear combination of the fundamental weights $\mu = m_1\lambda_1 +\cdots+ m_\ell\lambda_\ell$ where the coefficients $m_i = 2(\mu,\alpha_i)/(\alpha_i,\alpha_i)$ are called the Dynkin indices of the weight and are always integers. A weight λ belongs to S if and only if its Dynkin indices are nonnegative integers. The simple roots can also be written in terms of the fundamental weights as

$$\alpha_j = \sum_i \lambda_i A_{ij},$$

where

$$A_{ij} = 2(\alpha_i,\alpha_j)/(\alpha_i,\alpha_i)$$

are the entries in the Cartan matrix $A = [A_{ij}]$.

The linear transformation w_{α_i} from H_R^* to itself is defined by

$$w_{\alpha_i}(\mu) = \mu - (2(\mu,\alpha_i)/(\alpha_i,\alpha_i))\alpha_i$$

for $\mu\in H_R^*$. If μ is a weight of a representation ρ, then $w_{\alpha_i}(\mu)$ is also a weight of ρ, and its multiplicity is the same as that of μ. The linear transformation w_{α_i} is a reflection in the plane perpendicular to the root α_i and is called a Weyl reflection; the Weyl reflections w_{α_i}, $1 \le i \le \ell$, generate a a group called the Weyl group. It is now clear that the weight system and the multiplicity of a weight are invariant under an element of the Weyl group. It should also be noted that each element of the Weyl group is an orthogonal linear transformation so that its determinant is ± 1. Moreover, a presentation of the Weyl group is: generators $w_{\alpha_1},\cdots,w_{\alpha_\ell}$ and relations $(w_{\alpha_i}w_{\alpha_j})^{p_{ij}}$ = identity, where

$$p_{ij} = \begin{cases} 1 & \text{if } i = j \\ 2 & \text{if } i \neq j \text{ and } (\alpha_i,\alpha_j) = \pi/2 \\ 3 & \text{if } i \neq j \text{ and } (\alpha_i,\alpha_j) = 2\pi/3 \\ 4 & \text{if } i \neq j \text{ and } (\alpha_i,\alpha_j) = 3\pi/4. \\ 6 & \text{if } i \neq j \text{ and } (\alpha_i,\alpha_j) = 5\pi/6. \end{cases}$$

The Weyl group enables us to define the notion of equivalent weights. Two weights μ and ν are equivalent if there exists an element w in the Weyl group such that $w(\mu) = \nu$. A weight μ in a weight system is dominant if its Dynkin indices are nonnegative. It can be shown that any weight in a weight system is equivalent to exactly one dominant weight.

Example 2. Consider the simple Lie algebra B_2. There are two simple roots α_1 and α_2 and the Coxeter-Dynkin diagram is

The Cartan matrix is

$$A = \begin{bmatrix} 2 & -1 \\ -2 & 2 \end{bmatrix}.$$

If λ_1 and λ_2 are the fundamental weights, then $\alpha_1 = 2\lambda_1 - 2\lambda_2$ and $\alpha_2 = -\lambda_1 + 2\lambda_2$. The Weyl group of B_2 and a compact notation for it is:

identity	
w_{α_1}	1
$w_{\alpha_2} w_{\alpha_1}$	21
$w_{\alpha_1} w_{\alpha_2} w_{\alpha_1}$	121
w_{α_2}	2
$w_{\alpha_1} w_{\alpha_2}$	12
$w_{\alpha_2} w_{\alpha_1} w_{\alpha_2}$	212
$w_{\alpha_1} w_{\alpha_2} w_{\alpha_1} w_{\alpha_2}$	1212

3. Computational Representation Theory

A. Calculation of Inner Multiplicities

A number of different methods have been developed for the computation of inner multiplicities, and many calculations of these multiplicities have been carried out by hand for low ranks. In this section we discuss the methods due to Freudenthal, Kostant and Racah from a computational point of view. The first computer program for the computation of inner multiplicities was written by Agrawala and Belinfante [1]. It implemented Freudenthal's formula. This formula has also been implemented by Krusemeyer [13] and Patera and Sankoff [17]. Beck and Kolman have implemented Kostant's and Racah's formulas in addition to Freudenthal's for the purpose of examining computational efficiency [10].

All the above programs were written in Fortran except Krusemeyer's which was written in Algol 60.

Let μ be a weight in the weight system of an irreducible representation ρ with highest weight λ of a complex simple Lie algebra L of rank ℓ. Let $\alpha_1, \alpha_2, \cdots, \alpha_\ell$ be the simple roots of L, and let $\lambda_1, \lambda_2, \cdots, \lambda_\ell$ be the fundamental weights of L. Freudenthal's formula [6]

$$\{(\lambda + \delta, \lambda + \delta) - (\mu + \delta, \mu + \delta)\} n_\mu = 2 \sum_{\alpha > 0} \sum_{j=1}^{\infty} n_{\mu + j\alpha} (\mu + j\alpha, \alpha) \qquad (5)$$

provides a recursive method for computing the inner multiplicity of the weight μ in terms of the inner multiplicities of the higher weights. Here

$$\delta = \frac{1}{2} \sum_{\alpha > 0} \alpha = \sum_{i=1}^{\ell} \lambda_i$$

is half the sum of the positive roots, and (η, ξ) denotes the inner product induced by the Killing form. To use (5) to compute n_μ we start out with n_λ which we know is 1. Suppose now that μ is a weight other than the highest weight. We may write

$$\mu = \lambda - \sum_{i=1}^{\ell} k_i \alpha_i,$$

where the k_i are nonnegative integers. The <u>level</u> of μ is defined to be $\sum_{i=1}^{\ell} k_i$. Suppose further than we already know the multiplicities for weights $\mu' > \mu$ that is, for weights which can be written as

$$\mu' = \lambda - \sum_{i=1}^{\ell} k_i' \alpha_i,$$

where the k_i' are integers satisfying $0 \leq k_i' \leq k_i$. Then each term on the right side of (5) is known. One can show that for any weight $\mu \neq \lambda$, we have $(\mu + \delta, \mu + \delta) < (\lambda + \delta, \lambda + \delta)$. Hence, we can solve for n_μ as

$$n_\mu = \frac{2 \sum_{\alpha > 0} \sum_{j=1}^{\infty} n_{\mu + j\alpha} (\mu + j\alpha, \alpha)}{(\lambda + \delta, \lambda + \delta) - (\mu + \delta, \mu + \delta)} \qquad (6)$$

If $(\lambda + \delta, \lambda + \delta) - (\mu + \delta, \mu + \delta) = 0$, then μ is not a weight and we defined

$n_\mu = 0$. The sum on j is actually finite because there is an N such that if $j > N$ the vectors $\mu + j\alpha$ are higher than the highest weight and consequently $n_{\mu + j\alpha} = 0$.

Racah's formula [18] (also developed by Klymyk [9]) provides another recursive expression for n_μ. It is

$$n_\mu = - \sum_{\substack{w \in W \\ w \neq \text{identity}}} (\det w) n_{\mu + \delta - w(\delta)},$$

where W is the Weyl group associated with L. Note that $\delta - w(\delta)$ is a sum of simple roots with nonnegative integer coefficients [8, p. 248]. Consequently the inner multiplicity of $\mu + \delta - w(\delta)$ has already been computed if the same procedure as with Freudenthal's formula is used here.

Kostant's formula [12] involves a sum of values of a partition function; the sum is indexed by the Weyl group. The amount of computation necessary to evaluate the partition function makes the formula inefficient when compared to Freudenthal's or Racah's formulas.

To compute the inner multiplicities of the irreducible representation ρ with highest weight λ, we first must compute the weight system. The weight system is obtained by using the Dynkin algorithm [5]. The inner multiplicities are computed with Freudenthal's formula or Racah's formula.

The efficiency of the Freudenthal formula is due primarily to the Dominant Weight Lemma and the Layer Lemma which are given below.

Dominant Weight Lemma [13]

Let μ be a weight of ρ and let $\hat{\mu}$ denote its equivalent dominant weight. Define weights $\mu^{(n)}$, $n = 0,1,2,\cdots$, recursively as follows:

$$\mu^{(0)} = \mu,$$

$$\mu^{(1)} = \begin{cases} w_{\alpha_i}(\mu) & \text{if } (\mu,\alpha_i) < 0 \text{ and } (\mu,\alpha_j) \geq 0 \text{ for all } j<i, \\ \mu & \text{if } (\mu,\alpha_i) \geq 0 \text{ for all } i,\ 1 \leq i \leq \ell, \end{cases}$$

$$\mu^{(n)} = (\mu^{(n-1)})^{(1)}.$$

Then

(a) $\mu = \mu^{(1)}$ if and only if μ is dominant;

(b) for n large enough, $\mu^{(n)} = \hat{\mu}$.

This lemma says that starting with any weight μ we can arrive at the equivalent dominant weight $\hat{\mu}$ by a sequence of Weyl reflections.

Layer Lemma [2]

Let $\mu = \sum_{i=1}^{\ell} m_i \lambda_i$ be a weight of ρ. Suppose μ belongs to layer r. Then $w_{\alpha_j}(\mu)$ belongs to layer $r + m_j$.

Given a representation of a complex simple Lie algebra specified by the Dynkin indices of its highest weight λ, we can compute the inner multiplicities of all the weights or of just the dominant weights. The former procedure uses the Layer Lemma; the latter uses the Dominant Weight Lemma.

Racah's formula also depends on the Dominant Weight Lemma. To use the formula we first compute a partial table (indexed by part of the Weyl group) of $\delta - w(\delta)$ including enough entries so that the lowest dominant weight added to the table entry of highest level is greater than λ. It is not necessary to sum over the entire Weyl group nor to generate a listing of any part of it. These ideas will be discussed further in Section 3B. Then for any dominant weight μ, the Dominant Weight Lemma can be used to find $n_{\mu + \delta - w(\delta)}$ for any

table entry $\delta - w(\delta)$.

Computational experience has shown that Racah's formula is faster than Freudenthal's formula for small ranks due to the small number of summands. For large ranks Freudenthal's formula is more efficient.

B. Calculation of Outer Multiplicity

If (ρ',V') and (ρ'',V'') are irreducible representations of a complex semisimple Lie algebra L, then $(\rho' \otimes \rho'', V' \otimes V'')$ is a completely reducible representation of L defined by $(\rho' \otimes \rho'')(x)(v' \otimes v'') = (\rho'(x)v') \otimes v'' + v' \otimes \rho''(x)v''$. Since $\rho' \otimes \rho''$ is completely reducible, we may write

$$\rho' \otimes \rho'' = \rho_1 + \rho_2 + \cdots + \rho_k \qquad (8)$$

where $\rho_i (1 \leq i \leq k)$ is an irreducible representation of L. The ρ_i may not all be distinct. The number of times a given representation ρ appears in (8) is defined as the outer multiplicity of ρ in $\rho' \otimes \rho''$ and is denoted by $m^{\lambda}_{\lambda',\lambda''}$ where the representation ρ,ρ',ρ'' has highest weight $\lambda,\lambda',\lambda''$, respectively.

There have been a variety of formulas developed which can be used for computing outer multiplicities. Some demand knowledge of inner multiplicities, others start from girdles, and yet others require no preliminary calculations of other quantities from representation theory. The first computer implementations of several of these methods were done by Beck and Kolman [11]. These implementations were carried out in Fortran.

The Racah formula for the outer multiplicity of ρ in $\rho' \otimes \rho''$ is

$$m^{\lambda}_{\lambda',\lambda''} = \sum_{w \in W} (\det w)\, n^{\lambda'}_{\lambda + \delta - w(\lambda''+\delta)},$$

where W and δ are as before and the inner multiplicities are computed with respect to the representation whose highest weight is λ'.

Let

$$\chi'(h) = \sum_{\eta \in \Lambda(\rho')} n_\mu \exp(\mu(h))$$

$$\chi''(h) = \sum_{\eta \in \Lambda(\rho'')} n_\eta \exp(\eta(h))$$

be the characters of ρ' and ρ'' respectively. It can be shown [8, p.259] that the character of $\rho' \otimes \rho''$ is

$$\chi_{\rho' \otimes \rho''}(h) = \chi'(h)\chi''(h)$$

$$= \sum_{\mu,\eta} n_\mu n_\eta \exp((\mu + \eta)(h))$$

$$= \sum_{\gamma \in \Lambda(\rho' \otimes \rho'')} n_\gamma \exp(\gamma(h)).$$

By comparing these two expressions for the character of $\rho' \otimes \rho''$, we can deduce the following properties of the tensor product representation

A. Every weight of $\rho' \otimes \rho''$ is the sum of a weight of ρ' and one of ρ''; conversely, every such sum is a weight of $\rho' \otimes \rho''$.

B. The inner multiplicity of the weight λ in $\rho' \otimes \rho''$ is $n_\lambda = \sum_{\lambda=\mu+\eta} n_\mu n_\eta$

Let λ' be the highest weight of ρ and λ'' be the highest weight of ρ''. Then from properties A and B it follows that:

C. The highest weight of $\rho' \otimes \rho''$ is $\lambda' + \lambda''$.

D. The inner multiplicity of $\lambda' + \lambda''$ in $\rho' \otimes \rho''$ is 1.

Following directly from the statements about the character of a tensor product representation is a method of computing outer multiplicities which is sometimes called the method of successive subtractions. The weight system of $\rho' \otimes \rho''$ is formed as described in A and B. From C and D it follows that the highest irreducible component of $\rho' \otimes \rho''$ is the representation with highest weight $\lambda' + \lambda''$ and its outer multiplicity is one. Furthermore, the set of weights of $\lambda' + \lambda''$ is contained in the set of weights of $\rho' \otimes \rho''$. If the inner multiplicity of each weight in $\lambda' + \lambda''$ is subtracted from its inner multiplicity in $\rho' \otimes \rho''$, the

contribution of the irreducible piece $\lambda' + \lambda''$ is removed from $\rho' \otimes \rho''$. Among the remaining weights of $\rho' \otimes \rho''$ with nonzero inner multiplicity there is at least one highest dominant weight, call it λ, with respect to the order determined by the level of the weight system for $\rho' \otimes \rho''$. Again if the inner multiplicity of each weight in $\Lambda(\lambda)$ is subtracted $m^{\lambda}_{\lambda'\lambda''}$ times from its current inner multiplicity in $\rho' \otimes \rho''$, the contribution of the $m^{\lambda}_{\lambda'\lambda''}$ irreducible pieces ρ with the highest weight λ is removed from $\rho' \otimes \rho''$. This process of selecting the highest remaining dominant weight and removing its contribution to the weight system of $\rho' \otimes \rho''$ is repeated until the weight system of $\rho' \otimes \rho''$ is exhausted.

Steinberg's formula [19] for outer multiplicities involves a sum of values of the same partition function which occurs in Kostant's formula. This sum is indexed by the Weyl group. The amount of computation involved in this formula renders it inefficient when compared to Racah's formula.

The method of attack for using Racah's formula to compute outer multiplicities consists of first finding the dominant weights of the form $\lambda'' + \mu$, where $\mu \in \Lambda(\lambda')$. These are the only possible highest weights of irreducible components of $\rho' \otimes \rho''$. Then use Racah's formula to calculate the outer multiplicity of each of these weights. Similar to the Racah formula for inner multiplicities, we must generate a table of values of $\delta - w(\lambda'' + \delta)$. This table is generated recursively in order of increasing α-level, where the α-level of $\mu = \sum_{i=1}^{\ell} k_i \alpha_i$ is defined to be $\sum_{i=1}^{\ell} k_i$. The recursive method of generating the table is based on the following lemma.

Lemma

Assume the α-level of $\delta - w(\lambda'' + \delta) = \sum_{i=1}^{\ell} m_i \lambda_i$ is r. If $1 \leq j \leq \ell$, then the α-level of $\delta - w_{\alpha_j}(\lambda'' + \delta) = \sum_{i=1}^{\ell} (m_i + (1 - m_j)) a_{ij} \lambda_i$ is $r + 1 - m_j$, where $A = [a_{ij}]$ is the Cartan matrix.

If $\xi = \sum_{j=1}^{\ell} p_j \lambda_j$ already appears in the table, then for each j such that $p_j < 0$, we use the Lemma to find another table entry. Its determinant entry will be the negative of the determinant entry for ξ. This process may give duplications but they can easily be located since they will only occur within a given α-level. Since this table is used to compute elements of H_R^* whose inner multiplicities in the weight system of λ' are needed we can stop generating the table when

$$\alpha\text{-level}(\delta - w(\lambda'' + \delta)) \leq \alpha\text{-level}(\lambda').$$

A similar stopping procedure has been developed for the Racah inner multiplicity formula. These stopping conditions completely eliminate the necessity of having to generate the Weyl group to use Racah's formulas.

C. Calculation of Branching Multiplicities

Let (ρ, V) be an irreducible representation of the complex simple Lie algebra L. Let K be a subalgebra of L. By restricting ρ to K we obtain a representation ρ_K of K on V. If K is semisimple, then ρ_K is completely reducible. That is,

$$\rho_K = \rho_1 + \rho_2 + \cdots + \rho_r, \qquad (9)$$

where the ρ_i are irreducible representations of K and may not all be distinct. The number of times a given representation ρ' appears in (9) is defined as the branching multiplicity of ρ' in ρ_K and is denoted by $m^{\lambda'}_{\lambda}$, where λ and λ' are the highest weights of ρ and ρ' respectively. A program to calculate branching multiplicities has been implemented by

McKay, Patera, and Sankoff [16] in Fortran. This program deals only with the case where K is a maximal semisimple subalgebra of L. It uses an approach similar to the method of successive subtractions which was discussed in Section 3B.

4. Structural Analysis of a Lie Algebra

Much of Lie algebra has traditionally been existential in character. Several investigators [3, 15, 20] are currently involved in developing computational approaches to study the structure of a Lie algebra. Among the structural elements of a Lie algebra that one would like to calculate are:

(1) The nil and solvable radicals.
(2) The upper central, lower central, and derived series.
(3) The center.
(4) A Cartan subalgebra.
(5) The centralizer and the idealizer of a given subalgebra.

Algorithms to compute these and other structural elements have been developed by Beck, Kolman and Stewart [3]..

All of these algorithms assume that the algebra is given with a fixed basis $\{e_1,e_2,\ldots,e_n\}$ and structure constants $C = (c_{ij}^{k})$. A subalgebra or ideal spanned by $\{x_1,x_2,\ldots,x_r\}$ is specified by the $r \times n$ matrix $B = [b_{ij}]$, where $x_i = \sum_{j=1}^{n} b_{ij}e_j$. Two widely used algorithms from linear algebra are an algorithm to extend a set of linearly independent vectors to a basis and an algorithm to find a basis for the solution space of a homogeneous system of equations.

Suppose I is an r-dimensional ideal of the n-dimensional Lie algebra L, which is specified by the $r \times n$ matrix B, as above. The quotient algebra will be given by its structure constants $(\bar{c}_{pq}^{s})$ with respect to a basis $\{\bar{e}_1,\bar{e}_2,\ldots,\bar{e}_{n-r}\}$. The array $(\bar{c}_{pq}^{s})$ is computed by extending the basis for I to a basis for L, calculating the structure constants of L with respect to this new basis, and then selecting the appropriate subarray of this new structure constant array.

An n-tuple x representing an element of L is in the center of L if and only if x is in the solution space of the homogeneous system with coefficient matrix M given by the APL expression

```
M←(N×N,N←1↑⍴C)⍴C
```

The solvable radical can be readily computed as the orthogonal complement of [L,L] with respect to the Killing form assuming the characteristic of the ground field is 0. The algorithm to compute the solvable radical calls for first diagonalizing the Killing form restricted to [L,L], choosing an appropriate new basis for L, and then diagonalizing the Killing form again. The algorithm for calculating the nil radical is considerably more involved and can be found in [3].

Considerable attention has been devoted to the theoretical aspects of Cartan subslgebras. Their existence has not been fully settled for Lie algebras over finite fields. Cartan subalgebras over fields of characteristic zero can be obtained algorithmically, but the procedure is quite involved [3]. Certain elements of L, called regular elements, generate Cartan subalgebras. The set of regular elements of L is dense in L with respect to the Zariski topology on L. Thus, with high probability the random selection of an elements of L will yield a Cartan subalgebra.

5. External Structures

One of the algebraic structures which is studied for any nonassociative

algebra is its derivation algebra. This algebra is a Lie algebra of linear transformations. The derivation algebra can be obtained as the solution space of a certain homogeneous system.

Using the derivation algebra one can associate to any Lie algebra L another Lie algebra Hol(L) = D(L) + L, where D(L) is the derivation algebra of L, called the holomorph of L. An interesting type of problem is to discover conditions under which isomorphic holomorphs or isomorphic derivation algebras must have arisen from isomorphic Lie algebras. Leger and Luks [14] have considered such questions for the class of metabelian Lie algebras (L is metabelian if $L^3 = 0$ and $L^2 \neq 0$). Luks [15] has used the computer to develop examples showing that even in the restricted class of metabelian Lie algebras there are non-isomorphic Lie algebras whose holomorphs and derivation algebras are isomorphic. Luks has also used the computer to calculate certain cohomology groups of metabelian Lie algebras.

REFERENCES

[1] V.K. Agrawala and Johan G.F. Belinfante, Weight diagrams for Lie group representations: A computer implementation of Freudenthal's algorithm in ALGOL and FORTRAN, BIT 9(1969), 301-314.

[2] Robert E. Beck and Bernard Kolman, A computer implementation of Freudenthal's multiplicity formula, Indag. Math. 37(1972), 350-352.

[3] Robert E. Beck, Bernard Kolman and Ian Stewart, Computing the structure of a Lie algebra, in Computers in the Study of Nonassociative Rings and Algebras (Robert E. Beck and Bernard Kolman, Eds.) Academic Press, New York, to appear.

[4] J.G.F. Belinfante and Bernard Kolman, A Survey of Lie Groups and Lie Algebras with Applications and Computational Methods, SIAM, Philadelphia, 1972.

[5] E.B. Dynkin, Maximal subgroups of the classical groups. Supplement. Survey of the basic concepts and facts in the theory of linear representations of semisimple Lie algebras, Amer. Math. Soc. Transl. Ser. II 6(1957), 319-378.

[6] H. Freudenthal, Zur Berechnung der Charaktere der Halbeinfache Lieschen Gruppen, Indag. Math. 16(1954) 369-376.

[7] James E. Humphreys, Introduction to Lie Algebras and Representation Theory, Springer-Verlag, New York, 1972.

[8] N. Jacobson, Lie Algebras, Wiley-Interscience, New York, 1962.

[9] A.U. Klymyk, Decomposition of a direct product of irreducible representations of a semisimple Lie algebra into a direct sum of irreducible representations, Amer. Math. Soc. Transl. Ser. II 76(1968), 63-73.

[10] Bernard Kolman and Robert Beck, Computers in Lie algebras. I. Calculation of inner multiplicities, SIAM. J. Appl. Math. 25(1973), 300-312.

[11] Bernard Kolman and Robert E. Beck, Computers in Lie algebras. II. Calculation of outer multiplicities, SIAM. J. Appl. Math. 25(1973), 313-323.

[12] B. Kostant, A formula for the multiplicity of a weight. Trans. Amer. Math. Soc. 93 (1959), 53-73.

[13] M.I. Krusemeyer, Determining multiplicities of dominant weights in irreducible Lie algebra representations using a computer BIT 11(1971), 310-316.

[14] G. Leger and E. Luks, On derivations and holomorphs of nilpotent Lie algebras, Nagoya Math J. 44(1971), 39-50.

[15] E. Luks, Computing Lie algebra cohomology, in Computers in the Study of Nonassociative Rings and Algebras (Robert E. Beck and Bernard Kolman, Eds.) Academic Press, New York, to appear.

[16] W. McKay, J. Patera, and D. Sankoff, A computer program for calculating branching rules of representations of semisimple Lie algebras, in Computers in the Study of Nonassociative Rings and Algebras (Robert E. Beck and Bernard Kolman, Eds.) Academic Press, New York, to appear.

[17] J. Patera and D. Sankoff, Branching rules for representations of simple Lie algebras. Centre de Recherches Mathematiques, Universitè de Montrèal February 1972, unpublished.

[18] G. Racah, Lectures on Lie groups, in Group Theoretical Concepts and Methods in Elementary Particle Phy-

sics (F. Gursey, Ed.,), Gordon and Breach, New York, 1964.

[19] Robert Steinberg, A general Clebsch-Gordan theorem, Bull. Amer. Math. Soc. 67(1961), 406-407.

[20] Hans Zassenhaus, Subalgebras and invariants of Lie algebras, in Computers in the Study of Nonassociative Rings and Algebras (Robert E. Beck and Bernard Kolman, Eds.) Academic Press, New York, to appear.

Computational Techniques and the Structure of Groups in a Certain Class

C. M. Campbell

University of St. Andrews, St. Andrews, Scotland.

1. Introduction

The Todd-Coxeter algorithm enumerates the cosets of a finitely generated subgroup of finite index in a finitely presented group. The algorithm has been modified to give a presentation of the subgroup in terms of the given generators. In this paper we describe briefly computer programmes for the algorithm and the modified algorithm, and illustrate how the modified algorithm may be extended to subgroups of countable index.

We then describe some methods that enable us to obtain more information from a coset enumeration programme when space is limited. To illustrate these techniques we investigate a class of cyclically presented groups. We first state some theorems concerning this class. Using the computational techniques described earlier, we then discuss in some detail the structure of some of the particular groups involved.

The paper arises out of joint work with E. F. Robertson at the University of St. Andrews.

2. Computer Programmes

In this section we describe the two coset enumeration programmes used in our work. The first, by M. J. Beetham [1], has been implemented on the IBM 360/44 computer of the University of St. Andrews Computing Laboratory. It is able to determine an index up to 32,000/(2n+1) where n is the number of group generators.

Since the modified algorithm is not part of [1], we use a programme by N. W. G. Wilde [22], an IBM programme package entitled the 'Benson-Mendelsohn algorithm for certain word problems in groups'. The programme is in two parts. The first part is a coset enumeration programme. In addition, given a subgroup H of finite index in a group G then for each coset i and each group generator x_j the programme finds the following relationships involving cosets and a set of coset representatives

$$ix_j = k,$$

$$\tau_i x_j = w_{ij}\tau_k,$$

where k is a coset, τ_i and τ_k are coset representatives for cosets i and k respectively and w_{ij} is a word in the subgroup generators. The second part of the programme determines the words w_{ij} in the relationships $\tau_i x_j = w_{ij}\tau_k$. However it does not give a presentation for the subgroup H and so we wrote a programme CCRG to complete the modified algorithm. For output the line-printer prints out the relations for the subgroup. The punch provides card output for the relations of the subgroup in a form that can be used immediately by the programme [1]. The advantage of having relations printed as well as punched is that redundant relations may be discarded and relations for the subgroup may also be simplified. Thus in certain applications, see for example [8], the presentation for the subgroup is simplified before implementation of the programme [1]. We have also written another programme CCRGAB, a modification of CCRG, in which the relations for the subgroup are abelianised. Such a programme is useful if we know that the subgroup is abelian and may also be used to show that a group is infinite. Applications are given in [8] and in section 6.

3. The Algorithms

In [5] there is a description of the Todd-Coxeter algorithm in the form in which it has been implemented in [1]. The paper [2] also gives a description of the modified algorithm, versions of which are discussed by several other authors ([3], [4], [13], [16], [18], [20]). Of note is the fact that this modification to the algorithm enables us to find a presentation for a finitely generated subgroup of *countable* index in a finitely generated finitely presented group by using just one enumeration, other authors handling only finite index and that with two enumerations.

An application of this theorem is given in Theorem 4 of [8], where we show that the subgroup $H = \langle a, b^2 \rangle$ is of countable index in the group

$$G \cong \langle a,b \mid [a^2,b^{-1}] = a^{-1}b^{2n}a, [b^2,a^{-1}] = b^{-1}a^{2n}b \rangle.$$

Let $x = a$ and $y = b^2$ so that $H = \langle x,y \rangle$. Assuming that $n \geq 1$ we may obtain from the modified algorithm the following table (integers denote coset representatives):

$1a = x1$ $\quad 1b = 2$ $\quad 2b = y1$ $\quad 2a = 3$

$3a = xy^n x2$ $\quad 3b = 4$ $\quad 4b = yx^{2n}3$ $\quad 4a = 5$

$5a = xy^n xy^n 4$ $\quad 5b = 6$ $\quad 6b = yx^{2n}(xy^n x)^n 5$ $\quad 6a = 7$

...

and in general, for $k > 1$,

$$(2k-1)a = w(2k-1).(2k-2), \quad (2k-1)b = 2k,$$

$$(2k)b = w(2k).(2k-1), \quad (2k)a = 2k+1,$$

where $w(k)$ is given by $w(1) = x$, $w(2) = y$, $w(3) = xy^n x$, $w(4) = yx^{2n}$ and, by induction, $w(k) = w(k-2)[w(k-3)]^n$; $k \geq 5$.

Each coset gives two relations for the subgroup $\langle x,y \rangle$, one of the relations being trivial. Denote by $R(k) = 1$ the non-trivial relation obtained from the coset k. Then from the relation $ba^{2n}bab^{-2}a^{-1} = 1$ we get $R(1) = (xy^n x)^n yxy^{-1}x^{-1}$, and from the relation $ab^{2n}aba^{-2}b^{-1} = 1$ we get $R(2) = (yx^{2n})^n xy^n xyx^{-2}y^{-1}$. By induction, $R(k) = [w(k+2)]^n w(k+1)w(k)[w(k-1)]^{-1}[w(k)]^{-1}$; $k \geq 3$. Notice that if $n \leq 1$ the same argument holds. Thus we have the following theorem: if H is the subgroup $\langle a,b^2 \rangle$ of G, then $H \cong \langle x,y \mid R(k) = 1, k = 1,2,3,\ldots \rangle$ where the $R(k)$ are given inductively as above. That this is a presentation for H follows from Theorem 4 of [2].

In [8] we use the modified algorithm to show that

$$G \cong \langle a,b \mid [a^2,b^{-1}] = a^{-1}b^3a, [b^2,a^{-1}] = b^{-1}a^3b \rangle$$

is a group of order $2^{15}.3^3$. Such a use of the modified algorithm gives rise to the following observation. Although a larger machine than the IBM 360/44 may be able to determine the order directly, the method is nevertheless general. It would enable us to find the orders of finite groups of greater order that a larger machine could not itself handle directly.

We are primarily interested in cyclically presented groups, a term which we shall now define. A group G is said to be *cyclically presented* if

$$G \cong \langle a_1,a_2,\ldots,a_n \mid w_1 = 1, w_2 = 1, \ldots, w_n = 1 \rangle$$

where w_1 is a word in the generators $a_1,a_2,\ldots,a_n$ and w_{i+1} is obtained from w_i by applying the permutation $(1\ 2\ \ldots\ n)$ to the suffices of the a's.

The use of the modified Todd-Coxeter algorithm is not restricted to the study of particular groups. For example, it can be used to find the orders of the Fibonacci groups $\tilde{F}(r,n)$ whenever $r \equiv 1 \bmod n$, solving Problem 3 of [17]. This result and related results are proved in [6], [7] and [11].

The class of cyclically presented groups known to be finite and non-metacyclic is small. Examples are some of the groups given by Macdonald [19] and Wamsley [21], some examples in [8], [9], [10] and some examples given in section 6.

In this paper we shall take as an example to illustrate our techniques the class of groups $\tilde{G}(l,m,n)$ where

$$\tilde{G}(l,m,n) \cong \langle a,b \mid ab^m a^{-1}b^{-1}a^{-n}b^{1-l} = 1, \quad ba^m b^{-1}a^{-1}b^{-n}a^{1-l} = 1 \rangle.$$

The groups $\tilde{G}(m,m,n)$ are investigated in [8].

4. Computational methods

We now look at some results that we have used in conjunction with the coset enumeration programme [1].

Cyclic subgroups. Suppose we have a group G and suppose that coset enumeration shows that a word w_1 in the generators of G and their inverses generates a cyclic subgroup of index n. Further suppose that w_2 is another word in the generators of G and their inverses and that the index of $\langle w_1,w_2 \rangle$ in G is still n, then we know that w_2 belongs to the subgroup generated by w_1. Thus we may add to the presentation for G the relation that w_1 and w_2 commute, and similarly for any further words $w_3,w_4,\ldots,w_k$ that leave the index unaltered. The addition of such relations may, in certain cases, enable the coset enumeration programme [1] to determine the order of the group directly where, without these additional relations, this had previously proved impossible. Note that when we have a cyclic subgroup it is then particularly easy to use the modified algorithm.

Deduced relations. When we consider the group $\tilde{G}(2,3,-2)$ we show how, although the coset enumeration programme [1] is unable to determine the order of the group directly it may be used to determine relations which hold in the group and which simplify the coset enumeration. As well as using the computer to deduce that certain relations hold in the group, we may be able to find such relations algebraically.

Normal subgroups. If a subgroup H of a group G is generated by $h_1,h_2,\ldots,h_r$ where the h's are words in the generators of G and their inverses and if H has index n in G, then if on adding the relations $h_1 = 1$, $h_2 = 1,\ldots,h_r = 1$ to the presentation for G we obtain a group of order m, the index of the normal closure of H is m. Thus if $m = n$, H is a normal subgroup.

Commuting elements. If we wish to show that two elements w_1 and w_2 in a group G commute then we may add to the presentation for G the relation $[w_1,w_2] = 1$ and show that the order of the group is left unaltered.

Orders of words. An addition to the coset enumeration programme [1] enables us to determine the orders of words in a given group. Such information is particularly useful when we consider the Sylow structure of specific groups.

Other computational techniques that we have used, for example how we show that a Sylow subgroup has a particular structure, will be discussed as and when they arise in the examples to be considered in

section 6.

5. The groups $\underset{\sim}{G}(l,m,n)$

In this section we show how we investigated a certain class of groups. As our example we considered the groups

$$\underset{\sim}{G}(l,m,n) \cong \langle a,b \mid ab^m a^{-1} b^{-1} a^{-n} b^{1-l} = 1,\ ba^m b^{-1} a^{-1} b^{-n} a^{1-l} = 1 \rangle,$$

see also [5], [12]. We attempted, using [1], to determine the orders of the groups $\underset{\sim}{G}(l,m,n)$, $-2 \le l \le 5$, $-4 \le m, n \le 5$. From the tables of orders obtained it was obvious that there were isomorphisms between the groups, and the following theorem was then easily proved.

Theorem 1.

(1) $\underset{\sim}{G}(0,m,n) \cong \underset{\sim}{G}(0,-m,-n) \cong \underset{\sim}{G}(0,-n,-m) \cong \underset{\sim}{G}(0,n,m)$;

(2) $\underset{\sim}{G}(1,m,n) \cong \underset{\sim}{G}(1,1-n,1-m)$;

(3) $\underset{\sim}{G}(l,1,n) \cong \underset{\sim}{G}(l,n+l,1-l) \cong \underset{\sim}{G}(1-n,1,1-l) \cong \underset{\sim}{G}(1-n,2-l-n,n)$;

(4) $\underset{\sim}{G}(1-n,m,n) \cong \underset{\sim}{G}(2-n-m,2-m,m+n-1)$;

(5) $\underset{\sim}{G}(-n,-n,n+1) \cong \underset{\sim}{G}(1,1,n+1)$.

Other theorems on orders of the groups $\underset{\sim}{G}(l,m,n)$ then suggested themselves on the basis of the evidence in the tables. Sometimes we first wished to strengthen the evidence by computing the orders of certain specific groups outwith the tables. The following results were obtained.

Theorem 2. $\underset{\sim}{G}(l,m,n)/\underset{\sim}{G}'(l,m,n)$ is infinite if $m-l = \pm n$.

Theorem 3.

(1) $\underset{\sim}{G}(l,m,1) \cong \mathbb{Z}_{|m^2-2m|}$, the cyclic group of order $|m^2-2m|$;

(2) $\underset{\sim}{G}(1-n,2,n) \cong \mathbb{Z}_{|2n+1|}$;

(3) if $(l,n) = 1$, $\underset{\sim}{G}(l,0,n)$ has order l^2-n^2; otherwise $\underset{\sim}{G}(l,0,n)$ is infinite;

(4) $\underset{\sim}{G}(1,1,n)$ is a finite metacyclic group of order $|n|^3$. $\underset{\sim}{G}(1,1,-n)$ is isomorphic to $\underset{\sim}{G}(1,1,n)$.

Theorem 4. $\underset{\sim}{G}(l,m,0)$ is infinite if $(l,m) \neq 1$ and is metacyclic of order $|l-m|^3$ if $(l,m) = 1$.

Theorem 5. $G(l,m,n)$ is infinite if one of the following conditions holds:

(1) $m = -1$ and $(l-1,n-2) \ge 4$;

(2) $(l,m,n) \neq 1$;

(3) there exists an integer t such that $n \equiv 2 \bmod t$ and $(m+1,t) \ge 6$;

(4) there exists $t \ge 6$ with $m \equiv 1 \bmod t$, $n \equiv 1 \bmod t$ and $l \equiv 0 \bmod t$;

(5) there exists $t \ge 6$ with $m \equiv -1 \bmod t$, $n \equiv -1 \bmod t$ and $l \equiv 0 \bmod t$;

(6) $l = 1$, $n = 2$ and either $m \ge 3$ or $m \le -5$.

The proofs of parts (3) - (6) of Theorem 5 depend on the properties of (l,m,n) groups, see for example [14]. For a proof of Theorem 4 see [12]. Another question we may ask about the group $G(l,m,n)$ is to find conditions under which it has a simple homomorphic image. The following theorem may be obtained, see [5].

Theorem 6. Let p be a prime. Suppose $p \equiv 1 \bmod 4$ $i^2 \equiv -1 \bmod p$. Then $\underset{\sim}{G}(l,m,n)$ has $PSL(2,p)$ as a homomorphic image whenever

$$l \equiv 1+i \bmod p,\quad m \equiv 1+i \bmod p,\quad n \equiv -2i \bmod p. \quad (*)$$

If l,m and n satisfy $(*)$ and n is odd, then $\underset{\sim}{G}(l,m,n)$ has $SL(2,p)$ as a homomorphic image.

Finally in this section we consider the groups $\underset{\sim}{G}(1-n,m,n)$.

Theorem 7.

(1) If $n \equiv 0 \bmod (m-1)$, $\underset{\sim}{G}(1-n,m,n)$ is metacyclic of order $|(m-1)^2(2n+m-1)|$;

(2) if $n \equiv -1 \bmod (m-1)$, $\underset{\sim}{G}(1-n,m,n)$ is metacyclic of $|(2n+m-1)(m-1)(2^{m-1}-1)|$;

(3) if $n \equiv 1 \bmod (m-1)$, $\underset{\sim}{G}(1-n,m,n)$ has order

$$\begin{cases} 24\,|(2n+m-1)|(m-1)^2/(m+5)^2 & -4 \le m \le -1, \\ 24\,|(2n+m-1)|(m-1)^2/(m-7)^2 & 3 \le m \le 6, \end{cases}$$

and is infinite if $m \ge 7$ or $m \le -5$.

This theorem has the following corollary.

Corollary.

(1) If $n \equiv 1 \bmod 4$, $\underset{\sim}{G}(1-n,5,n)$ is finite and not metabelian;

(2) if $n \equiv 1 \bmod 5$, $\underset{\sim}{G}(1-n,6,n)$ is finite and not metabelian, having A_5 as a simple homomorphic image.

From this corollary we obtain two infinite series of cyclically presented finite non-metacyclic groups additional to the examples discussed in section 3.

6. The structure of groups

In this section we consider some groups $\underset{\sim}{G}(l,m,n)$ which are of special interest. In [8] it is shown that $\underset{\sim}{G}(2,2,3)$ and $\underset{\sim}{G}(2,2,-3)$ are finite non-metacyclic groups of orders $2^{15}.3^3$ and $2^8.3^3$ respectively. Here we show that the groups $\underset{\sim}{G}(2,3,-2)$, $\underset{\sim}{G}(-2,2,-1)$, $\underset{\sim}{G}(-2,3,-1)$ and $\underset{\sim}{G}(-1,-1,4)$ are also cyclically presented finite non-metacyclic groups.

$\underset{\sim}{G}(2,3,-2)$

$$\underset{\sim}{G}(2,3,-2) \cong \langle a,b \mid ab^3 a^{-1} b^{-1} a^2 b^{-1} = 1,\ ba^3 b^{-1} a^{-1} b^2 a^{-1} = 1 \rangle.$$

We shall show that $\tilde{G}(2,3,-2)$ is a group of order 1512. Although the coset enumeration programme [1] is unable to find the order of $\tilde{G}(2,3,-2)$ directly, it does give that the index of a is 252. It also gives that the index of the subgroup $\langle a,b^6 \rangle$ is 252 and so b^6 is in the subgroup generated by a. Therefore $[a,b^6] = 1$, and by symmetry $[b,a^6] = 1$. Also since $ab^3a^{-1}b^{-1}a^2b^{-1} = 1$ and $ba^3b^{-1}a^{-1}b^2a^{-1} = 1$,

$$b^5a^{-1}b^{-1}a^5b^{-1}a^{-1} = 1.$$

But $[a,b^6] = 1$ and $[b,a^6] = 1$ so

$$(ab)^3 = b^6a^6 = a^6b^6 = (ba)^3.$$

As in section 4, if we add to the presentation for $\tilde{G}(2,3,-2)$ the relations that we have deduced hold $\tilde{G}(2,3,-2)$, namely $[a,b^6] = 1$, $[b,a^6] = 1$, $(ab)^3 = (ba)^3$, then the coset enumeration programme [1] now gives $|\tilde{G}(2,3,-2)| = 1512$.

The Sylow 2-subgroup is $\mathbb{Z}_2 \times \mathbb{Z}_2 \times \mathbb{Z}_2$. For we may show that $b^{-1}a^3b^4a^{-1}b$, a^3 and $ba^2b^3a^{-2}b^{-1}$ are each words of order two, that any two of them generate a subgroup of order 4 and that the three of them generate the Sylow 2-subgroup. By adding on the relations $[b^{-1}a^3b^4a^{-1}b,a^3] = 1$, $[a^3,ba^2b^3a^{-2}b^{-1}] = 1$, and $[b^{-1}a^3b^4a^{-1}b,ba^2b^3a^{-2}b^{-1}] = 1$ to the presentation for $\tilde{G}(2,3,-2)$ we show, since the resulting group still has order 1512, that these three words of order 2, $b^{-1}a^3b^4a^{-1}b$, a^3 and $ba^2b^3a^{-2}b^{-1}$ commute. Therefore the Sylow 2-subgroup is $\mathbb{Z}_2 \times \mathbb{Z}_2 \times \mathbb{Z}_2$.

Further let H be the subgroup generated by ab^{-1} and $b^{-1}a$. Then from the modified Todd-Coxeter algorithm H is a normal subgroup of index 3 in $\tilde{G}(2,3,-2)$. With $x = ab^{-1}$ and $y = b^{-1}a$, CCRGAB gives the presentation $\langle x,y \mid x = y = 1 \rangle$ for H/H'. Thus H is perfect and since it has order 504 it is either PSL(2,8) the simple group of order 504 or has PSL(2,7) the simple group of order 168 as a homomorphic image. PSL(2,7) has Sylow 2-subgroup $\mathbb{Z}_8$. Thus $H \cong PSL(2,8)$ and $\tilde{G}(2,3,-2)$ is an extension of PSL(2,8) by $\mathbb{Z}_3$. $\tilde{G}(2,3,-2)$ is not soluble having a simple subgroup. It is probable that it is isomorphic to Edge's group of order 1512 [15] which also has PSL(2,8) as a normal subgroup of index three, but we have so far failed to show that the $\mathbb{Z}_3$ action is identical in both cases.

In a similar study of the groups of Fibonacci type we have discovered another two cyclically presented non-metacyclic groups of order 1512. These are the Fibonacci group $\tilde{F}(3,6)$ and the group $\tilde{F}(3,6,5,2)$ (see [9] and [10] respectively). It is straightforward to show that, unlike $\tilde{G}(2,3,-2)$, $\tilde{F}(3,6)$ and $\tilde{F}(3,6,5,2)$ are soluble, and then by considering the factor by the derived group for these groups that they are not isomorphic.

$\tilde{G}(-2,2,-1)$

Coset enumeration gives $|\tilde{G}(-2,2,-1)| = 1320 = 2^3.3.5.11$. The element a has order 30 and so a^{10} is an element of order 3. If we add on the relation $a^{10} = 1$ to the relations for $\tilde{G}(-2,2,-1)$ then we have a group of order 55 and so the Sylow 3-subgroups are not normal. Similarly if we add on the relation $a^6 = 1$, a^6 being an element of order 5, we find that this time the new group has order 24 and so the Sylow 5-subgroups are not normal. The element $(ab^{-1})^6$ is of order 11 and adding on the relation $(ab^{-1})^6 = 1$ we obtain a group of order 120 and so there is a unique normal Sylow 11-subgroup.

$(a^{-1}b^{-1}ab)^{11}$ and $(ab)^5$ are two elements of order 4 and together they generate the Sylow 2-subgroup. By adding on the relation $(ab)^5 = 1$ to those of $\tilde{G}(-2,2,-1)$ we obtain a group of order 168. Therefore, since the normal closure of $(ab)^5$ is of order 8, the Sylow 2-subgroup is normal. Since it is generated by two elements of order 4, and since it is not abelian, the Sylow 2-subgroup is not abelian.

$\tilde{G}(-2,2,-1)$ is not nilpotent since it is not the direct product of its Sylow subgroups. However, it is soluble since the derived group, being of order $2^3.11$, is of the form $p^\alpha q^\beta$ where p and q are primes.

$\tilde{G}(-2,3,-1)$

A similar procedure shows that $\tilde{G}(-2,3,-1)$, a group of order 5832, is not metabelian. Although soluble it is not nilpotent.

$\tilde{G}(-1,-1,4)$

Let H be the subgroup of $\tilde{G}(-1,-1,4)$ generated by a and b^2. Coset enumeration shows that H is a normal subgroup of index 2. Let $x = a$ and $y = b^2$. Then, using the modified algorithm,

$$H \cong \langle x,y \mid y^2x^{-3}yxy^{-1}x^{-2} = 1,\ y^2x^{-1}y^{-2}x^4y^2x = 1,\ x^{-1}y^{-2}xy^{-2}xy^{-2}x^{-2}y^3xy^{-1} = 1 \rangle.$$

The coset enumeration programme [1] now shows that $|H| = 3840 = 2^8.3.5$ and so $|\tilde{G}(-1,-1,4)| = 7680$. But by Theorem 6 $\tilde{G}(-1,-1,4)$ has $\tilde{PSL}(2,5)$ as a homomorphic image and it is therefore an extension of a group of order 2^7 by PSL(2,5). Since $\tilde{G}(-1,-1,4)$ has a simple homomorphic image it is not soluble and is therefore neither metacyclic nor nilpotent.

We now consider some other finite groups $\tilde{G}(l,m,n)$ that are of some computational interest. $\tilde{G}(2,3,-3)$ is a group of order 24. This order cannot be determined directly by the coset enumeration programme [1] but it does show that a has index 3 in $\tilde{G}(2,3,-3)$ as also does the subgroup $\langle a,b^2 \rangle$. As described in section 4, we may add the relation $[a,b^2] = 1$ and with this extra relation the order of $\tilde{G}(2,3,-3)$ is found to be 24. $\tilde{G}(2,3,-3)/\tilde{G}'(2,3,-3)$ is isomorphic to $\mathbb{Z}_8$ and so $\tilde{G}(2,3,-3)$ is the binary polyhedral group $\langle -2,2,3 \rangle$, see [14].

Now we turn to the group $\tilde{G}(-1,2,1)$. The subgroup generated by $\langle ab,ba \rangle$ has index 4 in $\tilde{G}(-1,2,1)$. With $x = ab$ and $y = ba$, the programme [22] and CCRG give as a presentation for H the

group generated by x and y subject to the six relations

$$xy^{-1}x^{-2}y^{-2}x^{-1}yx^2y^2 = 1,$$

$$xy^{-1}xy^2x^2yx^{-1}y = 1,$$

$$xy^2x^2yx^{-1}yx^2y^3x^{-1}y^2 = 1,$$

$$y^3x^2yx^{-1}yx^2yx^{-3}yxy^2x^2 = 1,$$

$$x^2y^{-2}x^{-2}y^{-1}xy^{-1}x^{-2}y^{-3}x^{-2}y^{-2} = 1,$$

$$y^{-3}x^{-2}y^3x^2 = 1.$$

From these relations we can show that

$$H \cong \langle x,y \mid x^6 = 1,\ x^3y^3 = 1,\ (xy^{-1})^2(x^{-1}y)^2 = 1 \rangle .$$

Coset enumeration now shows that H is of order 144 and therefore that $G(-1,2,1)$ is a group of order 576.

We now show that $G(0,3,-2)$ is a metabelian group of order 600. $|G(0,3,-2):G'(0,3,-2)| = 5$ and $G'(0,3,-2)$ may be generated by $[a^{-1},b^{-1}]$ and $[a^{-1},b]$. The metabelian property follows on showing that these two commutators commute.

In [8] it is shown that $G(2,2,2)$ and $G(2,2,-2)$ are isomorphic to D_∞, the infinite dihedral group, and that $G(2,2,4)$ is an extension of a group of order 32 by D_∞. Here we use the Wilde programme [22] and the programmes CCRG and CCRGAB, see section 2, to show that $G(4,-1,-1)$ and $G(4,-3,1)$ are infinite.

Coset enumeration gives $|G(4,-1,-1): \langle ab,ba \rangle| = 24$. Using the Wilde programme and the programme CCRG we find that if $x = ab$ and $y = ba$ then the subgroup $\langle x,y \rangle$ is the free group on x and y. Therefore $G(4,-1,-1)$ is infinite. Next for $G(4,-3,1)$ the subgroup $H = \langle a^2,b \rangle$ has index 3. The Wilde programme and CCRGAB then show that the factor by the derived group H/H' is given by

$$H/H' \cong \langle x,y \mid x^{16} = y^{16} \rangle$$

where $x = a^2$ and $y = b$. This implies that H is infinite and therefore that $G(4,-3,1)$ is infinite.

7. Tables

We give tables showing, where known, the orders of the groups $G(l,m,n)$, $-2 \le l \le 5$, $-4 \le m, n \le 5$. If there are two entries in a box then the upper half of the box gives the order of the group and the lower half one of the theorems, for there may be several, from which it has been deduced. If a box contains just one entry then that entry gives the order of the group and has been determined directly by the coset enumeration programme [1].

References

1. M.J. Beetham, "A programme for the Todd-Coxeter coset enumeration algorithm", (unpublished).
2. M.J. Beetham and C.M. Campbell, "A note on the Todd-Coxeter coset enumeration algorithm", Proc. Edinburgh Math. Soc. 20(1976), 73-79.
3. C.T. Benson and N.S. Mendelsohn, "A calculus for a certain class of word problems in groups", J. Combinatorial Theory 1 (1966), 202-208.
4. C.M. Campbell, "Enumeration of cosets and solutions of some word problems in groups", Dissertation, McGill University, 1965.
5. C.M. Campbell, "Applications of the Todd-Coxeter coset enumeration algorithm", Ph.D. thesis, University of St. Andrews, 1975.
6. C.M. Campbell and E.F. Robertson, "The orders of certain metacyclic groups", Bull. London Math. Soc. 6 (1974), 312-314.
7. C.M. Campbell and E.F. Robertson, "On metacyclic Fibonacci groups", Proc. Edinburgh Math. Soc. 19 (1975), 253-256.
8. C.M. Campbell and E.F. Robertson, "Remarks on a class of 2-generator groups of deficiency zero", J. Australian Math. Soc. 19 (1975), 297-305.
9. C.M. Campbell and E.F. Robertson, "Applications of the Todd-Coxeter algorithm to generalised Fibonacci groups", Proc. Roy. Soc. Edinburgh 73A (1974/75), 163-166.
10. C.M. Campbell and E.F. Robertson, "A note on Fibonacci type groups", Canad. Math. Bull. 18 (1975), 173-175.
11. C.M. Campbell and E. F. Robertson, "On a class of finitely presented groups of Fibonacci type", J. London Math. Soc. 11 (1975), 249-255.
12. C.M. Campbell and E.F. Robertson, "On a group presentation due to Fox", Canad. Math. Bull. (to appear).
13. J.J. Cannon, L.A. Dimino, G. Havas and J.M. Watson, "Implementation and analysis of the Todd-Coxeter algorithm", Mathematics of Computation 27 (1973), 463-490.
14. H.S.M. Coxeter and W.O.J. Moser, Generators and Relations for Discrete Groups (Springer, Berlin, 3rd ed. 1972).
15. W.L. Edge, "An operand for a group of order 1512", J. London Math. Soc. 7 (1973), 101-110.
16. G. Havas, "A Reidemeister-Schreier program", in Proc. Second Internat. Conf. Theory of Groups, Canberra 1973, edited by M.F. Newman (Springer, Berlin, 1974).
17. D.L. Johnson, J.W. Wamsley and D. Wright, "The Fibonacci groups", Proc. London Math. Soc. 29 (1974), 577-592.
18. J. Leech, "Computer proof of relations in groups", in Proceedings of the Royal Irish Academy Summer School on Group Theory and Computation (Academic Press, to appear).

19. I.D. Macdonald, "On a class of finitely presented groups", Canad. J. Math. 14 (1962), 602-613.

20. N.S. Mendelsohn, "Defining relations for subgroups of finite index of groups with a finite presentation", in Computational Problems in Abstract Algebra, edited by J. Leech (Pergamon, Oxford, 1970).

21. J.W. Wamsley, "A class of three generator three relation finite groups", Canad. J. Math. 22 (1970), 36-40.

22. N.W.G. Wilde, "Benson Mendelsohn algorithm for certain word problems in groups", IBM Contributed Program Library, No. 42.0.001 (1967).

$\underset{\sim}{G}(-2,m,n)$

n \ m	-4	-3	-2	-1	0	1	2	3	4	5
5					21 3.3	48 1.3		∞ 2		
4	∞ 5.2		∞ 5.2		∞ 5.2	7 1.3	∞ 5.2		∞ 5.2	
3		120 7.2	27 1.5	24 7.2	5 3.3	∞ 2	7 3.2	48 7.3	81 7.1	600 7.2
2	∞ 5.2		∞ 5.2	648	∞ 5.2	5 1.3	∞ 5.2		∞ 5.2	
1		∞ 2	120	∞ 2	3 3.3	24 1.3				
0	∞ 5.2	1 4	∞ 2	1 4	∞ 5.2	27 4	∞ 5.2	125 4	∞ 5.2	343 4
-1		∞ 2	1	∞ 2	3 3.3	120 1.3	1320	5832	1015	
-2	∞ 5.2		∞ 5.2	3	∞ 2		∞ 5.2		∞ 5.2	
-3					5 3.3	∞ 2				
-4	∞ 5.2		∞ 5.2		∞ 3.3		∞ 2		∞ 5.2	

$\underset{\sim}{G}(-1,m,n)$

n \ m	-4	-3	-2	-1	0	1	2	3	4	5
5				125	24 3.3	147 1.3			∞ 2	
4				7680	15 3.3	24 1.3		∞ 2		
3	∞ 2			27	8 3.3	5 1.3	∞ 2			
2	20	∞ 2	21 7.2	8 1.5	3 3.3	∞ 2	5 3.2	24 7.1	147 7.2	2048
1		3	∞ 2	1	∞ 3.3	3 1.3	576			
0	27 4	8 4	1 4	∞ 2	1 3.3	8 4	27 4	64 4	125 4	216 4
-1		3	∞ 2	120	∞ 3.3	21 1.3	40	165		
-2		∞ 2	3	8	3 3.3	∞ 2	5			
-3	∞			27	8 3.3	55	∞ 2			
-4					15 3.3			∞ 2		

$\underset{\sim}{G}(0,m,n)$

n \ m	-4	-3	-2	-1	0	1	2	3	4	5
5				∞ 1.1	∞ 1.1	576 1.1				∞ 5.2
4	∞ 1.1	192 1.1	∞ 1.1	1800 1.1	∞ 1.1	120 1.1	∞ 1.1		∞ 5.2	
3	192 1.1	∞ 1.1	600 1.1	192 1.1	∞ 1.1	24 1.1	5 1.1	∞ 5.2		
2	∞ 1.1	600 1.1	∞ 1.1	24 1.1	∞ 1.1	3 1.1	∞ 5.2	5	∞ 5.2	
1	1800 1.1	192 1.1	24 1.1	∞ 1.1	1 1.1	∞ 5.2	3	24 7.3	120 7.3	576 7.3
0	∞ 1.1	∞ 1.1	∞ 1.1	1 1.1	∞ 4	1 4	∞ 4	∞ 4	∞ 4	∞ 4
-1	120 1.1	24 1.1	3 1.1	∞ 1.1	1 2	∞ 2	24 7.2	192 1.3	1800 7.2	∞ 7.2
-2	∞ 1.1	5 1.1	∞ 1.1	3 1.1	∞ 1.1	24 1.1	∞ 5.2	600	∞ 5.2	
-3		∞ 1.1	5 1.1	24 1.1	∞ 1.1	192 1.1	600 1.1	∞ 5.2		
-4	∞ 1.1		∞ 1.1	120 1.1	∞ 1.1	1800 1.1	∞ 1.1	192 1.1	∞ 5.2	

$\underset{\sim}{G}(1,m,n)$

n \ m	-4	-3	-2	-1	0	1	2	3	4	5
5	∞ 2	171	144	168	24 1.2	125 4	192		16	
4	171 1.2	∞ 2	56	24	15 1.2	64 4	960	24	7	∞ 2
3	144 1.2	56 1.2	∞ 2	5	8 1.2	27 4	8	5	∞ 2	7
2	168 1.2	24 1.2	5 1.2	∞ 2	3 1.2	8 4	24	∞ 5.6	∞ 5.6	∞ 5.6
1	24 1.2	15 1.2	8 1.2	3 1.2	∞ 3.1	1 3.1	∞ 3.1	3 3.1	8 3.1	15 3.1
0	125 1.2	64 1.2	27 1.2	8 1.2	1 1.2	∞ 4	1 4	8 4	27 4	64 4
-1	192 1.2	960 1.2	8 1.2	24 1.2	∞ 1.2	1 3.4	∞ 2	3	8	120
-2		24 1.2	5 1.2	∞ 1.2	3 1.2	8 1.2	3 1.2	∞ 2	5	24
-3	16 1.2	7 1.2	∞ 1.2	∞ 1.2	8 1.2	27 1.2	8 1.2	5 1.2	∞ 2	56
-4		∞ 1.2	7 1.2	∞ 1.2	15 1.2	64 1.2	120 1.2	24 1.2	56 1.2	∞ 2

$\underset{\sim}{G}(2,m,n)$

n \ m	-4	-3	-2	-1	0	1	2	3	4	5
5		∞ 2		1220	21 3.3	1512 1.3	125			
4	∞ 5.2		∞ 5.2	301	∞ 3.3	465 1.3	∞ 5.2		∞ 5.2	
3		240	497	∞ 2	5 3.3	120 1.3	884,736			∞ 2
2	∞ 5.2	1344	∞ 5.2	55	∞ 3.3	21 2	∞ 5.2	3	∞ 5.2	
1	280	216	165	24	3 3.3	∞ 2	1	∞ 2	3	24
0	∞ 5.2	125 4	∞ 5.2	27 4	∞ 3.3	1 5.2	∞ 4	1 5.2	∞ 4	27 5.2
-1	1085 7.2	360 7.2	105 7.2	24 7.2	3 3.3	∞ 2	1 3.2	∞ 2	21 7.2	120 7.2
-2	∞ 5.2		∞ 5.2	5	∞ 3.3	3 1.3	∞ 5.2	1512	∞ 5.2	
-3		48	7	∞ 2	5 3.3	24 1.3	6912	24		∞ 2
-4	∞ 5.2		∞ 5.2	7	∞ 3.3	105 1.3	∞ 5.2		∞ 5.2	

$\underset{\sim}{G}(3,m,n)$

n \ m	-4	-3	-2	-1	0	1	2	3	4	5
5			∞ 2		16 3.3					
4				∞ 2	7 3.3					
3		∞ 5.2		497	∞ 3.3			∞ 5.2		
2			609		5 3.3	∞ 2	24			∞ 2
1	1200	1015		165	8 3.3	24 1.3	∞ 2	120	∞ 2	
0	343 4	8 4	125 4	64 4	∞ 4	8 4	1 4	∞ 4	1 4	8 4
-1		385		120	8 3.3	3 1.3	∞ 2	1	∞ 2	
-2		2048	168 7.3	24 7.1	5 3.3	∞ 2	3 3.2	8 7.1	24 7.3	∞ 2
-3		∞ 5.2		7	∞ 3.3	5 1.3		∞ 5.2		
-4				∞ 2	7 3.3	24 1.3				

$\underset{\sim}{G}(4,m,n)$

n \ m	-4	-3	-2	-1	0	1	2	3	4	5
5				∞ 2	9 3.3					
4	∞ 5.2		∞ 5.2		∞ 5.2		∞ 5.2		∞ 5.2	
3					7 3.3	∞ 2.6				
2	∞ 5.2		∞ 5.2		∞ 5.2	55	∞ 5.2		∞ 5.2	
1		∞	1015	216	15 3.3	192 1.3		∞ 2		∞ 2
0	∞ 5.2	343 4	∞ 5.2	125 4	∞ 5.2	27 4	∞ 5.2	1 4	∞ 5.2	1 4
-1		432	385	∞	15 3.3	24 1.3		∞ 2		∞ 2
-2	∞ 5.2		∞ 5.2	21	∞ 5.2	5 1.3	∞ 5.2		∞ 5.2	
-3		960 7.3	81 7.1	48 7.2	7 3.3	∞ 1.3	5 3.2	24 7.3	27 1.6	192 7.3
-4	∞ 5.2		∞ 5.2		∞ 5.2	7 1.3	∞ 5.2		∞ 5.2	

$\underset{\sim}{G}(5,m,n)$

n \ m	-4	-3	-2	-1	0	1	2	3	4	5
5					∞ 3.3					∞ 5.2
4					9 3.3	∞ 2				
3					16 3.3		∞ 2			
2					21 3.3			∞ 2		
1			1200	280	24 3.3	1800 1.3			∞ 2	
0	729 4	512 4	343 4	216 4	∞ 4	64 4	27 4	8 4	1 4	∞ 4
-1		1827		4200	24 3.3	105			∞ 2	
-2				∞ 5.1	21 3.3	24		∞ 2		
-3					16 3.3	7 2	∞ 2			
-4		192	231	40	9 3.3	∞ 2	7 3.2	24	105	64 1.5

The Computation and Application of Vector Congruence Classes

Zavdi L. Lichtman

Department of Mathematical Sciences

Tel-Aviv University

Tel-Aviv, Israel.

ABSTRACT

In a similar fashion to the partitioning of the integers into residue classes, integer spaces (finitely generated free Z-modules) can be partitioned into complete or incomplete vector congruence classes.

Complete vector congruence classes are computable by an application of the Smith normal form. The problem of computing efficiently incomplete vector congruence classes is discussed and a few methods are suggested.

The application of complete and incomplete vector congruence classes is in performing set-theoretic operations on integer spaces. All set-theoretic operations can be performed on integer spaces which partition into complete vector congruence classes. Some set-theoretic operations can also be performed on integer spaces which partition into incomplete vector congruence classes, but not always in a satisfactory manner.

1. INTRODUCTION

The problem of performing set-theoretic operations on integer spaces (finitely generated free Z-modules) arose in the design [8] of a new decision method for a subset of the first order theory of the integers with equality and addition (there are some restrictions on the arrangement of quantifiers). The aim was to utilize, as much as possible, the underlying algebraic structure and not just elementary number theory as is commonly done (see for example Cooper [7]).

Integer spaces can be partitioned into complete or incomplete vector congruence classes, and these structures provide the capabilities to perform set-theoretic operations.

Results in linear algebra over the integers, which form the basis for the definition and computation of vector congruence classes, were published more than a century ago by H.J.S. Smith [1].

Some properties of integer matrices are described in section 2. In section 3, Smith's results, which were given in the context of system of linear diophantine equations, are restated in the context of integer spaces, followed by the computation of vector congruence classes. Incomplete vector congruence classes are described in section 4, followed by a discussion of their computation in section 5. Set-theoretic operations are described in section 6 and some computational aspects are discussed in section 7.

2. INTEGER MATRICES

Definition 2.1. The greatest common divisor (gcd) of a rectangular integer matrix M is the gcd of the determinants of the largest square submatrices of M. In particular, the gcd of a square matrix is (the absolute value of) its determinant.

Definition 2.2. A square integer matrix M is a unimodular matrix iff $\det(M)=\pm 1$.

Unimodular matrices, and only these, have integer inverses [5,6] and unimodular linear operators are bijections between integer spaces.

The following are the three elementary operations on integer matrices [5,6]:
1. Interchange any two rows (columns).
2. Multiply a row (column) by ± 1.
3. Add an integer multiple of one row (column) to another.

Elementary row operations can be accomplished by multiplying the given matrix on the left by a unimodular matrix. Elementary column operations can be accomplished by multiplying the given matrix on the right by a unimodular matrix. These unimodular matrices, called elementary matrices, are obtained by performing the required elementary operations on the identity matrix. Every unimodular matrix is a product of a finite number of elementary matrices [5,6]. Two $m \times n$ integer matrices, M and N, are equivalent iff

there are two unimodular matrices, U and V, $m \times m$ and $n \times n$ respectively, such that $N = UMV$.

The rank of an integer matrix is its rank over the rationals. A matrix is said to have full row rank when its rank is equal to the number of rows.

Theorem 2.1 (Smith [1]). Every rectangular integer matrix M of rank r is equivalent to a matrix in Smith normal form

$$\begin{bmatrix} L & 0 \\ 0 & 0 \end{bmatrix}$$

where L is an $r \times r$ diagonal matrix with entries $d_1,\ldots,d_r$ $d_i > 0$, such that

$$d_1 | d_2,\ldots,d_{r-1} | d_r \text{ and } d_i = \frac{D_i}{D_{i-1}}$$

where D_i is the gcd of the determinants of the square submatrices (of M) of order i, and $D_o = 1$.

Given an integer matrix M , its Smith normal form S is unique. But in general there are infinitely many pairs of unimodular matrices U and V such that $S = UMV$ [2, p.392].

The principles of the computation of the Smith normal form are given by Mac Duffe [5], MacLane and Birkhoff [6], Newman [3], and of course by Smith [1]. A step by step algorithm is given by Bradley [4] .

3. VECTOR CONGRUENCE CLASSES

Definition 3.1. A residue vector $w \pmod D$ is an integer vector $w = (w_1,\ldots,w_m)$ such that $0 \leq w_i < D$ for all i .

Definition 3.2. A vector congruence class represented by a residue vector $w \pmod D$ is the set of all vectors v such that $v_i \equiv w_i \pmod D$ for all i .
Example: $(12,61,-2,3) \equiv (0,1,4,3) \pmod 6$.

Theorem 3.1 (Smith [1]). Let A be an $m \times n$ integer matrix with full row rank, and b an integer m-vector. The system of equations $Ax = b$ has an integer solution iff $\gcd(A) = \gcd([A;b])$, i.e. the coefficient and augmented matrices have the same gcd.

Corollary. Suppose $\gcd(A) = D$, and $Ax = b$ has an integer solution. If $b' \equiv b \pmod D$, i.e. b' and b belong to the same vector congruence class modulo D , then $Ax = b'$ also has an integer solution.

Proof. We have to show that $\gcd([A;b']) = D$. Consider the development of the determinants that include b in $[A;b]$ and b' in $[A;b']$ by expansion on b and b' respectively. The corresponding determinants can differ only by a multiple of D, and D divides every such determinant in $[A;b]$. Hence D divides every such determinant in $[A;b']$, so it divides the $\gcd([A;b'])$. Since $\gcd(A)=D$, $\gcd([A;b'])$ cannot be greater than D and hence $\gcd([A;b']) = D$.

Observation. Let A be an integer matrix. The set of all integer vectors b such that $Ax = b$ has an integer solution is the Z-module spanned by the column vectors of A .

Definition 3.3. We shall call a set of integer vectors (not necessarily independent) which spans a Z-module and arranged as the columns of a matrix a generating matrix for that Z-module. In particular, a basis of a Z-module, so arranged, is a generating matrix.

We can now restate theorem 3.1 and its corollary in the context of Z-modules.

Theorem 3.1 (restated). Let A be an $m \times n$ generating matrix with full row rank. An integer m-vector b is in the Z-module spanned by A iff $\gcd(A) = \gcd([A;b])$.

Corollary (restated). Let A be a generating matrix with full row rank and $\gcd(A) = D$, and let b be a vector in the Z-module spanned by A. If $b' \equiv b \pmod D$, then b' is also in the Z-module spanned by A.

Observation. There are D^m different m-vector congruence classes modulo D. This is immediate when we consider the residue m-vectors modulo D which represent these vector congruence classes. Each of the m entries in a residue vector can take all the D values $0,1,\ldots,D-1$.

The following theorem was given by Smith [1] in the context of systems of linear diophantine equations. It is given here directly in the context of Z-modules. The proof is supplied since it is the basis for the computation that follows.

Theorem 3.2. Let A be an $m \times n$ generating matrix with full row rank and $\gcd(A) = D$. The matrix A spans a Z-module which is a union of D^{m-1} vector congruence classes modulo D. $D^{m-1} \cdot (D-1)$ vector congruence classes modulo D are not spanned by A.

Proof. Let U and V, $m \times m$ and $n \times n$ respectively, be two unimodular matrices such that

$UAV = S$, S in Smith normal form and hence

$$UA = SV^{-1}$$

Since U is a unimodular linear operator, UA spans the same number of vector congruence classes modulo D as A does. Hence, SV^{-1} spans the same number of vector congruence classes modulo D as A does. Since for all vectors y $Sy = (SV^{-1})(Vy)$ and for all vectors z $(SV^{-1})z = S(V^{-1}z)$, S and SV^{-1} span the same Z-module and it is sufficient to count the number of vector congruence classes spanned by S.

It is clearly sufficient to count the number of residue vectors spanned by S, since by the (restated) corollary to theorem 3.1 a vector congruence class is spanned by S iff the residue vector representing it is spanned by S.

Let $d_1 = \frac{D_1}{D_0}$ $d_2 = \frac{D_2}{D_1}, \ldots, d_m = \frac{D_m}{D_{m-1}}$ be the diagonal entries of S (according to theorem 2.1), so $D_0 = 1$ and $D_m = D$, and $\prod_{i=1}^{m} d_i = D$. Each element w_i of a residue vector modulo D spanned by S can have D/d_i different values, therefore S spans

$$\prod_{i=1}^{m} \frac{D}{d_i} = \frac{D^m}{D} = D^{m-1}$$

different residue vectors modulo D.

The vector congruence classes spanned by a generating matrix with full row rank are directly computable by an application of the Smith normal form. The following example illustrates the computation.

Example. Let $A = \begin{bmatrix} -2 & 2 & 2 \\ 2 & 2 & 6 \end{bmatrix}$.

Since $\gcd(A) = 8$ and $8^{2-1} = 8$, A spans eight vector congruence classes modulo 8.

Two unimodular matrices which enable to obtain S, the Smith normal form of A, are:

$$U = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \quad \text{and} \quad V = \begin{bmatrix} -1 & 1 & -1 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{bmatrix}$$

giving:

$$S = UAV = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 4 & 0 \end{bmatrix}$$

The residue vectors determined by S can have in the first entry 0,2,4,6. These are all the multiples of 2 (the appropriate entry in S) which are less than 8 (the modulus). Likewise, the residue vectors can have 0 or 4 in the second entry. Thus, the matrix of residue vectors spanned by S is:

$$\begin{bmatrix} 0 & 2 & 4 & 6 & 0 & 2 & 4 & 6 \\ 0 & 0 & 0 & 0 & 4 & 4 & 4 & 4 \end{bmatrix}.$$

Premultiplying by $U^{-1} = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$:

$$\begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 & 2 & 4 & 6 & 0 & 2 & 4 & 6 \\ 0 & 0 & 0 & 0 & 4 & 4 & 4 & 4 \end{bmatrix} =$$

$$\begin{bmatrix} 0 & 2 & 4 & 6 & 0 & 2 & 4 & 6 \\ 0 & -2 & -4 & -6 & 4 & 2 & 0 & -2 \end{bmatrix},$$

and making all vectors residue vectors modulo 8:

$$\begin{bmatrix} 0 & 2 & 4 & 6 & 0 & 2 & 4 & 6 \\ 0 & 6 & 4 & 2 & 4 & 2 & 0 & 6 \end{bmatrix}.$$

The above computation is based on the proof of theorem 3.2. We first computed the vector congruence classes spanned by S. These are the same as the vector congruence classes spanned by SV^{-1}, which in turn are the same as those spanned by UA. Then, to obtain the vector congruence classes spanned by A we premultiplied by U^{-1}.

4. INCOMPLETE VECTOR CONGRUENCE CLASSES

A generating matrix which spans a Z-module may not have full row rank. Such a Z-module is a union of incomplete vector congruence classes. An incomplete vector congruence class is represented by a residue vector together with a set of rational linear forms indicating which vectors of the vector congruence class are in the incomplete vector congruence class.

We proceed to investigate the number of incomplete vector congruence classes in such Z-modules, and their concrete representation.

Let A be an $m \times n$ integer matrix of rank $r < m$, let $A_1, \ldots, A_m$ be the rows of A, and suppose these rows have been arranged so that $A_1, \ldots, A_r$ are linearly independent. Let A' be the $r \times n$ submatrix consisting of the first r rows of A,

and A'' the $(m-r) \times n$ submatrix consisting of the remaining rows of A. Then

$$A = \begin{bmatrix} A' \\ A'' \end{bmatrix} ,$$

A' has full row rank, and every row of A'' is a rational linear combination of the rows of A'. So for each i, $1 \leq i \leq m-r$, there is a rational r-vector f_i such that

$$A_{r+i} = f_i A' .$$

Let F be the rational $(m-r) \times r$ matrix

$$\begin{bmatrix} f_1 \\ \cdot \\ \cdot \\ \cdot \\ f_{m-r} \end{bmatrix}$$

then

$$A'' = FA' .$$

Theorem 4.1. Let A be an $m \times n$ generating matrix of rank $r < m$ and A' an $r \times n$ submatrix of A with full row rank, and $\gcd(A') = D$. The matrix A spans a Z-module which is a union of k incomplete vector congruence classes modulo D, where $D^{r-1} \leq k \leq D^{m-1}$.

Proof. Without loss of generality, we may assume that A' consists of the first r rows of A, so that

$$A = \begin{bmatrix} A' \\ A'' \end{bmatrix}$$

as in the preceding discussion.

By theorem 3.2 A' spans a Z-module of r-vectors with D^{r-1} complete vector congruence classes modulo D. For each integer r-vector v' in the span of A' there is a unique integer m-vector v in the span of A whose first r entries are these of v' (the remaining $m-r$ entries are Fv'), hence $k \geq D^{r-1}$.

For each i, $r < i \leq m$, the ith entry of a vector spanned by A can have at most D incongruent values modulo D. So the vector consisting of the last $m-r$ entries can have at most D^{m-r} incongruent values modulo D, and A spans at most $D^{r-1} \cdot D^{m-r} = D^{m-1}$ incongruent vectors modulo D, hence $k \leq D^{m-1}$.

Staying with the notation of theorem 4.1, the following is an explanation of that result. Let $v' = (v_1, \ldots, v_r)^T$ and $u' = (v_1, \ldots, v_{r-1}, v_r + D)^T$ be two vectors congruent modulo D spanned by A' and let F be the $(m-r) \times r$ rational matrix defined at the beginning of this section. Let $v'' = (v_{r+1}, \ldots, v_m)^T = Fv'$ and $u'' = (u_{r+1}, \ldots, u_m)^T = Fu'$. Then

$\begin{pmatrix} v' \\ v'' \end{pmatrix}$ and $\begin{pmatrix} u' \\ u'' \end{pmatrix}$ are both spanned by A and may or may not belong to the same vector congruence class modulo D, even though $v' \equiv u' \pmod{D}$.

The incompleteness of the vector congruence classes spanned by A follows from the fact that F determines a unique $(m-r)$-vector (spanned by A'') for every r-vector spanned by A'. If the vector $v = (v_1, \ldots, v_m)^T$ is spanned by A, then $(v_1, \ldots, v_{m-1}, v_m + D)^T$, which is congruent to v modulo D, is not spanned by A.

Incomplete vector congruence classes (spanned by a matrix of rank r) can be computed as follows: First compute the D^{r-1} vector congruence classes of r-vectors (spanned by a submatrix with full row rank). Then complete in all possible D incongruent ways each of the remaining $m-r$ entries (in each residue vector), allowing for all possible combinations.

However, some of these incomplete vector congruence classes might be empty. Let $F_o = [F; -I]$ where I is the $(m-r) \times (m-r)$ identity matrix. Then an integer vector v is spanned by A iff its first r entries are spanned by A' and $F_o v = 0$ (assuming that A is arranged as in theorem 4.1). And an incomplete vector congruence class represented by $w \pmod{D}$ and F is not empty (where w was constructed as described above) iff the system of equations

$$F_o(w + Dx) = 0$$

has an integer solution $x = (x_1, \ldots, x_m)$. This can be found by first obtaining a rational nonhomogeneous system of linear equations. Then, multiplying each equation by an appropriate integer, a system of linear diophantine equations is obtained. The solvability, in integers of this system is directly decidable (see for example Newman [3] or Bradley [4]).

5. ON THE COMPUTATION OF INCOMPLETE VECTOR CONGRUENCE CLASSES

The computation of non-empty incomplete vector congruence classes as described at the end of section 4 can be improved. In one case we can compute directly the non-empty incomplete vector congruence classes. In the other, less favorable, case we can reduce the set of potential non-empty incomplete vector con-

gruence classes, or even try a recursive procedure for a direct computation.

Consider a generating matrix A with less than full row rank, as at the beginning of section 4. Note that for every vector v' spanned by A' and for every rational r-vector f_i, $1 \le i \le m-r$, $f_i v'$ is an integer. In particular, if $\gcd(A') = D$ and $w' \pmod D$ is a residue vector spanned by A', then $f_i w'$ is an integer $(1 \le i \le m-r)$, and for all vectors v' such that $v' \equiv w' \pmod D$ $f_i v'$ is an integer $(1 \le i \le m-r)$. But this is possible iff each of the components of the rational r-vectors f_i can be represented as k/D for some integer k.

We have just proved the following theorem:

Theorem 5.1. Let A be an $m \times n$ generating matrix of rank $r < m$, A' an $r \times n$ submatrix of rank r and $f_1,\ldots,f_{m-r}$ the rational r-vectors describing the linear dependence on A' of the other $m-r$ rows. Then each of the entries in $f_1,\ldots,f_{m-r}$ can be represented as k/D for some integer k, where $D = \gcd(A')$.

This is also immediate when we consider each non-redundant system equivalent to $A' f_i = A_{r+i}$. Each has a single solution and by Cramer's rule all the solution components are of the form k/d where d is the determinant of the coefficient matrix. And our D is the gcd of all these determinants.

Let A be a generating matrix as in theorem 5.1 and F the $(m-r) \times r$ rational matrix whose rows are $f_1,\ldots,f_{m-r}$. Let $G = D \cdot F$ be the integer matrix obtained by multiplying each entry in F by D. We treat G as a generating matrix and suppose that F and G have full row rank. This is the first case in which we can compute directly the non-empty incomplete vector congruence classes.

Let $w' = (w_1,\ldots,w_r)$ be a residue vector modulo D spanned by A'. We wish to compute all the extensions $(w_{r+1},\ldots,w_m) \pmod D$ such that $(w_1,\ldots,w_m) \pmod D$ represents a non-empty incomplete vector congruence class spanned by A.

We compute a first extension by applying F to w'. Clearly, Fw' is spanned by A'' and therefore $Fw' \pmod D = (w_{r+1},\ldots,w_m) \pmod D$ is an extension of w', i.e. $(w_1,\ldots,w_m) \pmod D$ is a non-empty incomplete vector congruence class. All the vectors $v=(v_1,\ldots,v_m)$ spanned by A and such that $(v_1,\ldots,v_r) \equiv (w_1,\ldots,w_r) \pmod D$, are those in which $(v_{r+1},\ldots,v_m) = F \cdot (v_1,\ldots,v_r)$. But every such vector $(v_{r+1},\ldots,v_m)$ is the sum of Fw' and a vector spanned by G. Therefore, from the residue vector $(w_{r+1},\ldots,w_m) \pmod D$ and the vector congruence classes modulo $\gcd(G)$ spanned by G we can compute directly the non-empty incomplete vector congruence classes spanned by A. Let $E = \gcd(G)$ and $y_{(1)} \pmod E,\ldots, y_{(E^{m-r-1})} \pmod E$ be the residue vectors representing complete vector congruence classes spanned by G (we write $y_{(i)}$ for the ith vector and y_i for the i_{th} entry in a given vector y).

All possible entries w_{r+1} in the extensions of w' are computed as follows: For a residue vector $y \pmod E$ compute the sequence $(w_{r+1}+y_1+jE) \pmod D$ for $j = 0,1,\ldots$ until a repeat occurs. All the incongruent values modulo D (at most D of them) so obtained are all the possible values of the $r+1^{st}$ entry due to the particular $y \pmod E$. Using the same method we compute the $r+2^{nd}$ to m^{th} possible values of the $r+2^{nd}$ to m^{th} entries due to the particular $y \pmod E$. Then, the extensions of w' to w (due to $y \pmod E$) are all possible vectors $(w_{r+1},\ldots,w_m)$ constructed by taking all possible combinations of values in the $m-r$ entries (where values for each entry are taken from those computed above).

The above computation has to be repeated for each w' spanned by A' with all E^{m-r-1} residue vectors modulo E spanned by G. However, each of the sequences $(w_{r+i}+y_i+jE) \pmod D$ contains* one element when E is a multiple of D. In these cases, when $D = E$ each vector w' spanned by A' has E^{m-r-1} extensions, namely $((w_{r+1},\ldots,w_m)+y_{(i)}) \pmod D$ $i = 1,\ldots,E^{m-r-1}$. When $E = iD$ $i > 1$, we have to consider only $y_{(i)}$'s which are incongruent modulo D.

Example. Let

$$A = \begin{bmatrix} -2 & 2 & 2 \\ 2 & 2 & 6 \\ 1 & 3 & 7 \\ 0 & -2 & -4 \end{bmatrix}$$

rank $(A) = 2$ and let A' be composed of the top two rows (this is the same matrix of the example in section 3), so $D = \gcd(A') = 8$.

* This will happen often since the determinant of each $(m-r)\times(m-r)$ matrix in G is D^{m-r} times the corresponding determinant in F.

Clearly, $A_3=1/2A_1+A_2$ and $A_4=-1/2A_1-1/2A_2$ (this can be computed using Cramer's rule according to the remark following theorem 5.1). So we have

$$F=\begin{bmatrix} 1/2 & 1 \\ -1/2 & -1/2 \end{bmatrix} \text{ and } 8\cdot F=G=\begin{bmatrix} 4 & 8 \\ -4 & -4 \end{bmatrix}$$

G is a square matrix so gcd(G)=det(G)=16. Using the method developed in section 3 we can compute the residue vectors modulo 16 representing the vector congruence classes spanned by G. The $16^{2-1}=16$ residue vectors modulo 16 are

$$\begin{bmatrix} 0 & 4 & 8 & 12 & 0 & 4 & 8 & 12 & 0 & 4 & 8 & 12 & 0 & 4 & 8 & 12 \\ 0 & 12 & 8 & 4 & 4 & 0 & 12 & 8 & 8 & 4 & 0 & 12 & 12 & 8 & 4 & 0 \end{bmatrix}$$

but we are interested only in those that are incongruent modulo 8, namely

$$\begin{bmatrix} 0 & 4 & 0 & 4 \\ 0 & 4 & 4 & 0 \end{bmatrix}.$$

We shall choose a residue vector spanned by A' say w'=(6,6), and compute all its extensions. To obtain the initial extension we compute

Fw'(mod D) = (9, -6)(mod 8)
= (1,2)(mod 8).

Clearly, the four extensions to (6,6)(mod 8) are obtained by adding the four residue vectors computed above to (1,2)(mod 8). So the non-empty incomplete vector congruence classes starting with (6,6) are

$$\begin{bmatrix} 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 \\ 1 & 5 & 1 & 5 \\ 2 & 6 & 6 & 2 \end{bmatrix}.$$

In a similar way we can compute the non-empty incomplete vector congruence classes starting with the other seven residue vectors spanned by A'. Clearly, in this particular example the number of non-empty incomplete vector congruence classes is $8\cdot 4 = 32$.

The above computation is much faster than computing all the $8^3 = 512$ potential non-empty incomplete vector congruence classes and deleting the empty ones, as suggested at the end of section 4.

We now turn to the second, less favorable, case when F does not have full row rank. Note that theorem 5.1 also holds in this case, and let G=D·F.

The most natural thing to do seems to apply recursively the computation described above. The computation will converge because each rational matrix describing linear dependence has less rows than the matrix whose linear dependence it describes.

However, we might run into some problems, to see that suppose that the set of non-empty incomplete vector congruence classes modulo E spanned by G were computed. Here E = gcd(G') where G' is some submatrix of G with full row rank (same relation as A' to A), and suppose that E is not a multiple of D. Then we cannot compute the extensions of a w' spanned by A' as described above because G spans incomplete vector congruence classes, and this problem can occur at any level of the recursive computation. It is not quite clear how to overcome this problem when it occurs.

We always can apply partially the method described for the case when G has full row rank. We identify a G' which is a submatrix of G with full row rank and apply the method to G' extending w' in the entries corresponding to the rows of G'. For the other entries we can do better than filling each with all D incongruent values as at the end of section 4. Suppose row G_i was not included in G'. Let w_{r+i} be the corresponding entry in Fw'(mod D). The possible values for the $r+i^{th}$ entry are $(w_{r+i}+j\cdot\gcd(G_i))(\text{mod } D)$. We perform this computation for all entries w_{r+i} not corresponding to rows of G', and fill these entries with all combinations of the values computed. It should be noted that the procedure described above produces a set of potential non-empty incomplete vector congruence classes. But this set will generally be much smaller than the one produced as suggested at the end of section 4.

6. SET-THEORETIC OPERATIONS

It is possible to perform set-theoretic operations on integer spaces* when they are represented as unions of (complete or incomplete) vector congruence classes. But it is first required to obtain a common modulus for all the integer spaces involved. The obvious choice is the least common multiple of all the moduli.

Let $w = (w_1,\ldots,w_m)$ be a residue vector modulo D, and let D divide D'. The vector congruence class represented by w(mod D) is a union of $(\frac{D'}{D})^m$ vector congruence classes modulo D'. The residue vectors modulo D' representing these vector congruence classes can be obtained by adding to each entry

* of vectors of equal length, of course.

in w all multiples of D such that $w_i+nD < D'$ and allowing for all combinations.

If $w \pmod D$ represents an incomplete vector congruence class, then all the rational linear forms associated with it are also associated with each of the corresponding residue vectors modulo D'.

Integer spaces which are unions of complete vector congruence classes are applicable to all set theoretic operations (union, intersection, subtraction and complementation). These operations are simply performed on the vector congruence classes representing the integer spaces. It is also possible to check whether or not a union of such integer spaces is the complete integer space.

This is done by checking if the union includes all the D^m vector congruence classes.

The situation is more complicated when incomplete vector congruence classes are involved. Intersection and union can still be computed. When computing an intersection, the rational linear forms associated with both integer spaces are associated with incomplete vector congruence classes in the intersection. When computing a union, an incomplete vector congruence class that occurred in both integer spaces includes vectors that conform with either one of the two sets of rational linear forms.

It can be shown that the union of a finite set of incomplete vector congruence classes having the same residue vector $w \pmod D$ is a proper subset of the complete vector congruence class represented by $w \pmod D$.

7. COMPUTATIONAL ASPECTS

When computing residue vectors or performing set-theoretic operations on integer spaces, computational difficulties are expected when the modulus is large. Therefore, it is advantageous to have the smallest possible modulus. For example, when computing residue vectors from a generating matrix with less than full row rank, if the largest submatrix with full row rank is not unique then the one with the smallest gcd should be chosen (provided that an efficient method for making this choice can be devised). Thereby, deriving fewer incomplete vector congruence classes.

A more concise representation of residue vectors is readily available. A residue vector $w \pmod D$ can be represented as a single number (provided the modulus is known), obtained by evaluating the vector w in a number system of radix D. Alternatively, this number can be used as an index for a boolean vector with D^m entries (where m is the length of the vectors involved), this is especially appropriate when computing a union.

It is well known that the floating point representation (and computation) in computers is inaccurate. Therefore, the rational matrices describing the linear dependence of the rows of an integer matrix must be produced in rational form, i.e., each entry has to be produced as a pair of integers, using integer arithmetic. As we saw this can be done using Cramer's rule.

8. CONCLUSION (AND PROBLEMS TO BE SOLVED)

Computational methods for obtaining a representation of integer spaces by complete and incomplete vector congruence classes were given. The representation of integer spaces by (complete and incomplete) vector congruence classes provides the capabilities to perform set-theoretic operations on these spaces.

There are some, unsolved yet, problems concerning the computation and application of incomplete vector congruence classes.

Also, it is of interest to find a way to identify integer spaces (i.e. with closure property) which are not the complete integer space (i.e. integer spaces with gcd > 1) inside unions of complete vector congruence classes.

Solving these problems will make vector congruence classes a more attractive tool for the manipulation of integer spaces.

REFERENCES

1. Smith, H.J.S. On Systems of Linear Indeterminate Equations and Congruences. Phil. Trans. London 151 (1861), 293-326. (reprinted in [2]).
2. Smith H.J.S. Collected Mathematical Papers, Vol. I, Chelsea Pub. Co., N.Y. 1965.
3. Newman, M. Integral Matrices, Academic Press, N.Y., 1972.
4. Bradley, G.H. Algorithms for Hermite and Smith Normal Matrices and Linear Diophantine Equations, Mathematics of Computation 25 (October 1971), 897-907.
5. Mac Duffee, C.C. An Introduction to Abstract Algebra, Wiley, N.Y., 1959.
6. Mac Lane, S. and Birkhoff, C. Algebra, Macmillan, N.Y., 1968.
7. Cooper, D.C. Theorem Proving in Arithmetic without Multiplication. Machine Intelligence 7, Edinburgh Univ. Press 1972.
8. Lichtman, Z.L. An Algebraic Approach To Equality Logic Over the Integers Without Multiplication, Ph.D. thesis, Syracuse University, 1974.

Simplification of Radical Expressions

by

B. F. Caviness[1]
Computer Science Department
Illinois Institute of Technology
Chicago, Illinois

and

R. J. Fateman
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California at Berkeley
Berkeley, California

Abstract

In this paper we discuss the problem of simplifying unnested radical expressions. We describe an algorithm implemented in MACSYMA that simplifies radical expressions and then follow this description with a formal treatment of the problem. Theoretical computing times for some of the algorithms are briefly discussed as is related work of other authors.

1. Introduction

In this paper we discuss the problem of simplifying unnested radical expressions. By radical expressions we mean, roughly speaking, sums, differences, products, and quotients of rational functions raised to rational powers. Thus we will be discussing algorithms for carrying out transformations such as

$$\frac{x-y}{\sqrt{x}+\sqrt{y}} \to \sqrt{x} - \sqrt{y} \tag{1}$$

$$\frac{(x-1)^{1/4}}{(1-x)^{1/4}} - \frac{(x-1)^{3/4}}{(1-x)^{3/4}} - \sqrt{2} \to 0 \tag{2}$$

and $1/(x+\sqrt{2}+3^{2/3}) \to$

$$[((2x^3-4x-9)\sqrt{2} - x^4 - 9x + 4)3^{2/3} + ((-9x^2-6)\sqrt{2} + 3x^3 + 18x + 27)3^{1/3} + (-x^4+4x^2+18x-4)\sqrt{2} + x^5 - 4x^3 + 9x^2 + 4x + 18] / [x^5-6x^4+18x^3 + 12x^2 + 108x + 73]. \tag{3}$$

We will primarily consider only expressions in which the rational exponentiation is unnested, that is, like those expressions in (1), (2) and (3) above as opposed to expressions like $[(x+1)^{1/2}+(x-1)^{1/2}]^{1/2}$ in which the square root is nested.

As is the case with all simplification problems there are three desirable attributes for any solution. First it is desirable to have a clean theoretical solution to the problem. Second the algorithms to carry out the simplifications should be as efficient as possible, and finally the solution should have good human engineering features, i.e., the use and actions of the simplification procedures should be as natural and transparent as possible. Unfortunately these three aspects of a solution are not usually compatible with each other and compromises and choices must be made. This is certainly the case for the simplification of radical expressions. Standard techniques from the theory of algebraic field extensions can be used to get a nice theoretical solution to the problem. However the nice theoretical solution neither leads to fast algorithms nor possesses nice human engineering features.

The theoretical solution involves finding an algebraic extension field of $Q(x_1,\ldots,x_n)$, the field of rational functions in $x_1,\ldots,x_n$ with coefficients in Q the field of rational numbers, to which the given radical expressions belong. In the algebraic extension field each element can be represented uniquely and this unique representation can be considered to be the simplified form for the given expression. Thus for example

$$\frac{2^{1/4}+3^{1/2}[-(x^2+1)]^{1/3}}{(x^2+1)^{1/2}} \tag{4}$$

is a member of the field $Q(x)[\omega_6,2^{1/4},3^{1/2},(x^2+1)^{1/6}]$. Each element of this field may be uniquely represented in the form

$$\alpha_0+\alpha_1(x^2+1)^{1/6}+\ldots+\alpha_5(x^2+1)^{5/6} \tag{5}$$

where α_i is in $Q(x)[\omega_6, 2^{1/4}, 3^{1/2}]$. In particular (4) may be written in the form (5) and is thus

$$(\frac{1}{x^2+1})(2^{1/4})(x^2+1)^{3/6}+(\frac{1}{x^2+1})\omega_6(3^{1/2})(x^2+1)^{5/6}. \tag{6}$$

---

(1) Present address: Department of Mathematical Sciences, Rensselaer Polytechnic Institute, Troy, N.Y. 12181

However from a human engineering point of view (4) would usually be more desirable than (6). Furthermore, to find the extension field of Q(x) to which (4) belongs requires that one verify that the polynomials y_1^4-2 and y_2^2-3 are irreducible over $Q(x)[\omega_6]$ and $Q(x)[\omega_6, 2^{1/4}]$ respectively. In general such polynomials have to be factored over algebraic extension fields. Currently no efficient algorithms are known for factoring polynomials over algebraic extension fields although recently Wang [Wan 75] and Weinberger [Wei 76] have reported some progress on finding such algorithms.

In section 2 the MACSYMA algorithm for simplifying radical expressions will be presented. In particular the compromises that were made to try to balance the clean theoretical solution with the efficiency and human engineering problems will be discussed.

In simplifying radical expressions there are other problems which must be confronted. For example what does the symbol $\sqrt{x}$ mean? The square root function is multi-valued. Which branch, if any, does the symbol $\sqrt{x}$ represent? Also, what do symbols like $\sqrt{x^2}$ mean? Is this to be taken as -x, x or $|x|$? To answer these questions precisely requires a careful discussion as is given in section 3. Roughly speaking the answers are that $\sqrt{x}$ can stand for either of the single-valued branches of the square root function without affecting the way the arithmetic and simplification is carried out. The ambiguity in the symbol $\sqrt{x^2}$ is inherent and an arbitrary choice must be made to resolve it. The choice of $|x|$ for this symbol would lead to certain difficulties that we do not consider herein. In section 3 one precise way is given that changes $\sqrt{x^2}$ into x. With suitable modifications the same ideas could be used to change it into -x. Basically what is done in section 3 is to give a set of rules that transform ambiguous expressions into non-ambiguous ones. The rules given there are not the only ones that can be given but the important point is that the ambiguities must be resolved and the rules for resolving them should be clearly stated. Section 3 also contains a description of the algorithm for finding a common algebraic extension field to which a given set of radical expressions belongs.

In section 5 a brief discussion of theoretical computing times for some of the algorithms of section 2 and 3 is presented. Section 5 follows section 4 in which some theorems that show a priori the irreducibility of polynomials like y_1^4-2 and y_2^2-3 over certain algebraic extension fields. This knowledge simplifies and speeds-up the algorithms for finding the extension fields.

In section 6 an alternative approach to finding field extensions, based on the so-called theorem on the primitive element [vdW 49], is discussed. This section also contains a review of other related research.

2. The MACSYMA Algorithm RADCAN

In this section we present an informal description of the algorithm that is used in MACSYMA for the simplification of expressions containing radicals. The algorithm which is called RADCAN, an acronym for RADical CANonical simplifier, was implemented by Fateman [Mat 75] and was initially available in 1970. In our discussion we will concentrate on RADCAN's handling of radicals although the program has the capability to handle a larger class of functions. More information about RADCAN can be found in [Fat 72].

Programs with purposes similar to those of RADCAN have been written by Fitch [Fit 71 and Fit 73] for CAMAL, by Shtokhamer [Sht 75a and Sht 75b] for REDUCE, and by Jenks [GJY 75] for SCRATCHPAD.

RADCAN attempts to convert an arbitrary expression containing radicals, exponentials and logarithms into a simpler form. The result produced by RADCAN can be in various forms depending on which options are chosen. There are three options: (i) an option to produce the result in form 1 or form 2, (ii) an option to rationalize denominators, and (iii) an option to combine products of expressions raised to the same fractional power.

With the form 1 option, all polynomials appearing under radicals are factored into irreducibles, i.e., prime integers or polynomials irreducible over the integers. With the form 2 option, the polynomials appearing under radicals are factored into square-free, pair-wise relative prime factors.

For example with the form 1 option RADCAN transforms $(x^2-1)^{1/2}$ into $(x-1)^{1/2}(x+1)^{1/2}$ but with the form 2 option $(x^2-1)^{1/2}$ is not changed. The expression $(x^4+9x^3+29x^2+39x+18)^{1/2}$ is transformed into $(x+3)(x+1)^{1/2}(x+2)^{1/2}$ with the form 1 option and is transformed into $(x+3)(x^2+2x+2)^{1/2}$ with the form 2 option. $(x^4+9x^3+29x^2+39x+18)^{1/2} + (x+2)^{1/3}$ will be transformed into $(x+3)(x+1)^{1/2}(x+2)^{3/6} + (x+2)^{2/6}$ under either the form 1 or form 2 option. The dinstinction between form 1 and form 2 is discussed further in section 3.

With the rationalization of denominator option the expression $(x-y)/(\sqrt{x}-\sqrt{y})$ will be transformed into $\sqrt{x}+\sqrt{y}$. Otherwise it is not transformed. This option will also produce the transformation (3) above.

Option (iii) is used in the final step of RADCAN to determine whether or not expressions like $\sqrt{2}\ \sqrt{3}$ should be rewritten as $\sqrt{6}$.

Except for expressions involving roots of unity, RADCAN with the form 1 and rationalization of denominators option is a canonical form [Cav 70] for the unnested radical expressions.

With the form 2 option any unnested radical expression equivalent to zero which does not involve roots of unity will be transformed to zero. The form 2 option requires less computation than the form 1 option since finding square-free, pair-wise relatively prime factors for non-constant polynomials is much easier than finding irreducible factors.

Experience with MACSYMA has indicated that it is not generally desirable to automatically invoke the rationalization of denominators option because simple expressions like the left-hand side of (3) are frequently transformed into complicated expressions like the right-hand side of (3).

As will be explained later to reduce zero-equivalent expressions involving roots of unity to zero, requires in general the factorization of polynomials over algebraic number fields, a task for which no efficient algorithms are known. So one of the compromises made in RADCAN was not to handle algebraic relationships involving roots of unity in a completely general way to avoid such factorization problems. Thus RADCAN will not reduce $(-1)^{1/4} - (-1)^{3/4} - (2)^{1/2}$ to zero. Although this is an expression that is unlikely to occur in practice, roots of unity do get introduced in natural ways since, for example, RADCAN transforms $(-x+1)^{1/4}$ into $(-1)^{1/4}(x-1)^{1/4}$.

The RADCAN algorithm consists of three basic phases which, depending upon the expression being simplified, may have to be repeated. The first phase is a global pass through the expression collecting the set R of all subexpressions which are radicals, exponentials, or logarithms. From R is generated a basis B in terms of which the elements of R can be represented. If R consists of only radicals, B is generated by factoring the polynomials that appear under the radicals. The particular set B that is generated depends on which option, form 1 or form 2, is taken.

After B is generated, appropriate radicals of the elements of B are considered as new variables and the original expression is represented in terms of the new and original variables.

In the second phase standard rational function simplification procedures such as expansion, collection of terms over a common denominator, and removal of greatest common divisors are applied to the arguments of any imbedded non-rational functions as well as to the expression as a whole. In this phase the radicals are considered simply as new independent variables. All knowledge of algebraic dependence is submerged.

The third phase reimposes the interpretation of the variables as radicals. The transformation $(\alpha^{1/a})^b \rightarrow \alpha^q(\alpha^{1/a})^r$ is applied for each radical $\alpha^{1/a}$ with $b \geq a$ where $b = qa + r$, $0 \leq r < a$. Thus $(2^{1/6})^7$ would be changed to $2(2^{1/6})$. If this transformation is applied to any radical in the expression, the expression must be rationally simplified again. Furthermore if the expression contains nested radicals then the above transformation can change nested radicals to unnested ones. A contrived example that illustrates this possibility is

$(x^2 + 3x + 3 + (x+1)^{1/2}(x+1)^{1/2})^{1/2}$. After phase 1 and phase 2 the expression would be in the form $(x^2 + 3x + y^2 + 3)^{1/2}$ where $y = (x+1)^{1/2}$. The phase 3 transformation changes y^2 to $x+1$ and then the reapplication of phases 1 and 2 simplifies the original expression to $x+2$ after which no further changes are made.

If appropriate, phase 3 also carries out options (ii) and (iii), in this order.

The three phases of simplifying an expression S may be further subdivided as follows.

I. First Phase

1. Make a list R of all distinct radicals in the expression S. Collect the distinct radicands, i.e., polynomials appearing under radicals, on a list P.

2. For each radicand P_i on P

(i) If P_i is an integer, factor it completely, placing the distinct prime factors on a list B, the basis list. Only factors of P_i not already on B are placed on B. So B always contains at most one instance of an element.

(ii) If P_i is a polynomial of positive degree, factor it into factors irreducible over the integers for the form 1 option or into square-free factors for the form 2 option. For the form 1 option place all irreducible factors not occurring on B thereon. In the case of the form 2 option let P_i^* denote a square-free factor of P_i. If P_i^* is relatively prime to each element on B, place P_i^* on B. Otherwise there exist some P_j on B with $\gcd(P_i^*, P_j) = C \neq 1$. Remove P_j from B and replace it with the two factors C and (P_j/C). If $C = P_i^*$ then repeat the process on the next square-free factor of P_i. If $C \neq P_i^*$ then replace P_i^* by P_i^*/C and repeat the immediately preceding procedure for P_i^*.

At the completion of phase 2 B will be a list of polynomials of non-negative degree which are square-free and pair-wise relatively prime. Under the form 1 option each element on B will also be irreducible.

3. As step 2 processing takes place, each radical on the list R is constantly updated to reflect the changing representation of each of the original radicals in terms of the latest basis B.

4. Associated with each element on B is a radical degree. If, for example, $x+1$ is on B and the only occurrences of $x+1$ in the radicals on R are $(x+1)^{1/3}$ and $(x+1)^{2/5}$ then the radical degree of $x+1$ is $15 = \text{lcm}(3,5)$. As B changes in step 2 the radical degrees must be revised also. In the above example $(x+1)^{1/3}$ and $(x+1)^{2/5}$ are represented in terms of a new variable $v_j = (x+1)^{1/15}$ as v_j^5 and v_j^6 respectively. In the case the base is -1 and the radical degree is m, the distinguished indeterminate $\omega_m = e^{i\pi/m}$ is introduced.

II. Second Phase

At this point the expression S is considered as a rational function in its original variables and a set of new variables obtained from the elements on B and their radical degrees. S so considered is rationally simplified.

III. Third Phase

The algebraic properties of the new variables are reintroduced. If any reductions are achieved

each of the three phases must be repeated.

A well-known general procedure [Pol 50, p.38] for rationalizing denominators is based on a straightforward application of the extended Euclidean algorithm [Knu 69, p.377, exercise 3]. We will explain the algorithm by an example. To rationalize the denominator we merely need to find its inverse. So to rationalize the denominator in (3) we find the inverse of $x+\sqrt{2}+3^{2/3}$. We consider $x+\sqrt{2}+3^{2/3}$ as an element of the field $Q(x)[\sqrt{2}, 3^{1/3}]$ which is isomorphic to the polynomial ring $Q(x)[y_1,y_2]$ modulo the ideal generated by $y_1^2-2=0$ and $y_2^3-3=0$. Now y_1^2-2 is irreducible over $Q(x)$ and y_2^3-3 is irreducible over $Q(x)[\sqrt{2}]$. Thus we consider $x+\sqrt{2}+3^{2/3}$ as the polynomial $y_2^2+(x+\sqrt{2})$ in the variable y_2 with coefficients in $Q(x)[\sqrt{2}]$. We apply the extended Euclidean algorithm to $y_2^2+(x+\sqrt{2})$ and y_2^3-3 to find polynomials $S(y_2)$ and $T(y_2)$ with coefficients in $Q(x)[\sqrt{2}]$ such that

$$S(y_2)[y_2^2 + x + \sqrt{2}] + T(y_2)[y_2^3 - 3] = a \neq 0 \quad (7)$$

where a is also in $Q(x)[\sqrt{2}]$. We are guaranteed that a is in $Q(x)[\sqrt{2}]$ since y_2^3-3 is irreducible and hence cannot have a factor of positive degree in common with $y_2^2+x+\sqrt{2}$.

If we consider (7) modulo y_2^3-3 as we should since y_2 is just another symbol for $3^{1/3}$ and hence $y_2^3-3=0$, we have that $S(y_2)[y_2^2+x+\sqrt{2}] = a \neq 0$. Thus $\frac{1}{y_2^2+x+\sqrt{2}} = \frac{1}{a}S(y_2)$. Furthermore the S and T produced by the extended Euclidean algorithm will have degrees in y_2 at most 2 and 1 respectively. In this example $S(y_2) = -(x+\sqrt{2})^{-1}y_2^2 + 3(x^2+2\sqrt{2}x+2)^{-1}y_2 + 1$ and $a = x+\sqrt{2}+9/(x^2+2\sqrt{2}x+2)$. Thus

$$\frac{1}{x+\sqrt{2}+3^{2/3}} = -\left(\frac{x+\sqrt{2}}{D}\right)3^{2/3} + \left(\frac{3}{D}\right)3^{1/3} + \frac{x^2+2\sqrt{2}x+2}{D} \quad (8)$$

where $D = (x+\sqrt{2})^3 + 9 = (3x^2+2)\sqrt{2} + x^3 + 6x + 9$.

To obtain (3) we must rationalize D with respect to $\sqrt{2}$. To do this we again apply the extended Euclidean algorithm, this time, to y_1^2-2 and $D = (3x^2+2)y_1 + x^3 + 6x + 9$ considered as polynomials in $y_1 = \sqrt{2}$ over $Q(x)$. We find that $1/D = [-(3x^2+2)\sqrt{2} + x^3 + 6x + 9]/E$ where $E = x^6 - 6x^4 + 18x^3 + 12x^2 + 54x + 73$.

In general we must apply the extended Euclidean algorithm at most one time for each algebraic extension.

This algorithm for rationalizing denominators has been programmed for multiple algebraic number extensions in the MACSYMA algebraic arithmetic system by Richard Zippel and Barry Trager.

Examples of the use of RADCAN may be found in [MaF 71].

3. Theoretical Issues

In this section we formally define the concept of a radical expression in both a syntactic and semantic sense, give formal procedures for simplifying and performing arithmetic on radical expressions, and discuss the relationship between the formal presentation and the previously discussed algorithm RADCAN.

Radical expressions are rational roots of polynomials, rational functions, and other radical expressions. They are built up from:

(i) the field Q of rational numbers,
(ii) the variables $x_1, x_2, \ldots, x_n$,
and (iii) the operations of addition, subtraction, multiplication, division, composition, and rational exponentiation.

We will restrict our attention to unnested radical expressions, i.e., expressions like $((x^2+2x)^{1/2}+5)^{2/3}$ will not be considered further. Shtokhamer [Sht 75b] has recently discussed one way in which these ideas can be modified and extended to cover nested radical expressions.

Given a radical expression such as

$$((x^2-1)/(x+2))^{1/2}/(x+3x+2)^{1/3} \quad (9)$$

we wish to interpret it as a single-valued branch of a meromorphic function in an algebraic extension of the field $Q(x_1,\ldots,x_n)$ of multivariate rational functions over Q. Since in general there are a number of single-valued branches that might reasonably be associated with an expression like (9) we would like to interpret (9) in as general, but yet as natural, a way as possible.

There are two different aspects to the interpretation problem that are intimately related to the way that we simplify and perform computations with radical expressions. It is important that these two different aspects be clearly distinguished.

The first aspect of the problem is to take a set of radical expressions and to find an algebraic extension field to which they belong and to find the representation of the expressions in this extension field. The finding of the extension field and the representation of the expressions within the field is the process that is frequently referred to as "simplification of radical expressions," and this is essentially the process that is carried out by RADCAN.

The desired extension field is not uniquely determined by the symbols for the radical expressions. For example $(x^3)^{1/3}$ satisfies the polynomial $y^3-x^3=0$ which has the irreducible factors y-x and y^2+xy+x^2. To do the arithmetic in $Q(x)((x^3)^{1/3})$, i.e., the field of rational functions in x with $(x^3)^{1/3}$ adjoined, we must know the irreducible polynomial of which $(x^3)^{1/3}$ is a root and then perform arithmetic in the polynomial ring $Q(x)[y]$ modulo the ideal generated by the irreducible polynomial which $(x^3)^{1/3}$ satisfies. We have two choices for the extension field: namely $Q(x)[y]/(y-x)$ which is isomorphic to $Q(x)$ and $Q(x)[y]/(y^2+xy+x^2)$ which is not

isomorphic to Q(x).

Either of the choices is a legitimate choice. The first choice effectively gives us the simplification $(x^3)^{1/3} \rightarrow x$ whereas the second one gives $(x^3)^{2/3} \rightarrow -x(x^3)^{1/3} - x^2$ and the symbol $(x^3)^{1/3}$ would not be rewritten or simplified at all. Of course one would normally take the first interpretation in this case, but there are other cases that are not so obvious. For example should $(x^2)^{1/2}$ be considered as x or -x, i.e., should we choose y-x or y+x as the irreducible factor of $y^2 - x^2$?

So we conclude that there are choices to be made, no one of which is more right than the other, at least from a mathematical point of view. The important point is that the inherent ambiguity of these symbols must be realized and the rules chosen for the resolution of the ambiguities must be stated clearly and careful. Of course the rules may be chosen by the system designer or they may be left to the user. Below we describe the set of rules that are used by RADCAN to resolve the ambiguities. These rules attempt to give as natural an interpretation as possible to the symbols.

The second aspect of the interpretation problem is illustrated by the following simple example. Suppose we have resolved the previously discussed ambiguities and have found that our radical expressions belong to the field $Q(x)[\sqrt{x}]$ which is isomorphic to $Q(x)[y]/(y^2-x)$. What does $\sqrt{x}$ mean? From a general point of view, it is a multivalued function so when we evaluate it at $x = 4$, either +2 or -2 is a legitimate value. If we want to interpret it as a single-valued branch which branch do we choose and what difference does it make? The answer is that it does not make any difference which branch we choose when we are doing the field operations of addition, subtraction, multiplication, and division. That is, if we take two functions from $Q(x)[\sqrt{x}]$ such as $(\frac{1}{x+1})\sqrt{x} + 1$ and $(2x^2 - 2)\sqrt{x} + x$ and perform a field operation on them such as multiplication the answer will be $(\frac{2x^3 + 2x^2 - x - 2}{x+1})\sqrt{x} + 2x^2 - x$ no matter which single-valued branch we choose for the interpretation of $\sqrt{x}$.

Once we have chosen the algebraic field extension to which the expressions belong, there is no longer any ambiguity in carrying out the field operations. The answers depend on the symbols alone and not on the interpretations. When other operations such as evaluation and substitution are performed the expressions may become ambiguous again. We will have more to say about these two operations later in this section.

The foregoing should be considered as an introduction to the rigorous discussion that follows. We first give a set of rules for resolving the inherent ambiguities discussed previously along with a procedure for finding the field extensions to which a finite set of radical expressions belongs. So we assume we are given a finite set of radical expressions. The expressions are scanned and a set of radicals R is formed where each member of R is of the form $(P_1/P_2)^r$ with P_1 and P_2 relatively prime members of $Z[x_1,\ldots,x_n]$. Z denotes the ring of integers; r is a positive member of Q and $r \neq 1$. If $P_2 = 1$ then we write $(P_1)^r$. P_1 and P_2 are assumed to be relatively prime in the strong sense that their only common divisors in Z are ± 1. Henceforth we will always use relatively prime in this strong sense. Furthermore it is assumed that P_2 is positive, that is, that its leading coefficient is a positive member of Z. Let $\mathcal{P}$ consist of all polynomials P_1, P_2 where $(P_1/P_2)^r$ appears in R.

Following Collins [Col 74] we define a basis for the set $\mathcal{P}$ of polynomials of degree ≥ 0 to be a set $\mathcal{B}$ of positive, square-free elements of $Z[x_1,\ldots,x_n]$ satisfying the following three conditions:

(i) If B_1 and B_2 are distinct members of $\mathcal{B}$, they are relatively prime.
(ii) If B is in $\mathcal{B}$ then $B \mid P$ for some P in $\mathcal{P}$.
(iii) If P is in $\mathcal{P}$, there exist B_i in $\mathcal{B}$ and positive integers e_i such that $P = \pm \Pi B_i^{e_i}$.

Note that our definition of basis differs slightly from Collins'. His definition is only for a set $\mathcal{P}$ containing primitive polynomials of positive degree in x_n. In our definition the elements of $\mathcal{P}$ are neither required to be primitive nor to be of positive degree in any variable.

Let $\mathcal{B}$ be a basis for $\mathcal{P}$. In finding an algebraic extension of $Q(x_1,\ldots,x_n)$ to which each element $(P_1/P_2)^r$ of R belongs we impose the following rules to resolve the inherent ambiguities:

(i) $(P_1/P_2)^r = P_1^r/P_2^r$.
(ii) If P_1 (or P_2) $= +\Pi B_i^{e_i}$ where each B_i is in $\mathcal{B}$ and each e_i is a positive integer, then $P_1^r = \Pi B_i^{e_i r}$. If $P_1 = -\Pi B_i^{e_i}$, then $P_1^r = (-1)^r \Pi B_i^{e_i r}$.
(iii) Suppose $r = s/t$ where s,t are positive, relatively prime members of Z. If $s = qt + v$, $0 \leq v < t$, then $B^r = B^q B^{v/t}$ for B in $\mathcal{B}$. Also $(-1)^r = (-1)^q (-1)^{v/t}$.

These assumptions imply that $\sqrt{x^2} = \sqrt{(-x)^2} = (x^3)^{1/3} = x$ and similarly resolve other inherently ambiguous expressions.

Given R and assuming (i) - (iii) above, we find an extension field to which each member $(P_1/P_2)^r \in R$ belongs by the following procedure:

(a) Using (i) above, each element of R, $(P_1/P_2)^r$, can be written as a quotient of radical polynomials of the form P_1^r/P_2^r. P_1 and P_2 are called radicands. Form the set $\mathcal{P}$ of all distinct radicands.

(b) Compute a basis $\mathcal{B}$ for the set $\mathcal{P}$.

(c) Using the properties of a basis and (ii) and (iii) above, each P_1^r and P_2^r can be uniquely expressed as a product of integer and fractional powers of members of $\mathcal{B} \cup \{-1\}$.

(d) For each B in $\mathcal{B}$ let $s_1/t_1,\ldots,s_k/t_k$ be the reduced rational powers to which it appears

after the steps (a) - (c) are carried out for each member of R. Let $t = \&cm(t_1,\ldots,t_k)$. If $k = 0$, $t = 1$. Then every element of R belongs to the algebraic extension field F obtained by adjoining all the $B^{1/t}$ to $Q(x_1,\ldots,x_n)$.

For example if $R = \{(2/3)^{1/3},(x+1)^{1/2}, (2-2x^2)^{1/2}\}$ then $P = \{2,3,x+1,2x^2-2\}$, $B = \{2,3,x+1,x-1\}$ and $F = \{Q(x)((-1)^{1/2},2^{1/6},3^{1/3}, (x+1)^{1/2},(x-1)^{1/2})\}$.

To do the arithmetic in F in a canonical fashion we must know the minimal polynomial M(y) satisfied by each $B^{1/t}$. Of course M(y) will be a factor of y^t-B. In the example above we must know the irreducible factors of y^2+1 over $F_0 = Q(x)$, the irreducible factors of y^6-2 over $F_1 = F_0(i)$, the irreducible factors of y^3-3 over $F_2 = F_1(2^{1/6})$, the irreducible factors of $y^2 - (x+1)$ over $F_3 = F_2(3^{1/3})$, and the irreducible factors of $y^2 - (x-1)$ over $F_4 = F_3((x+1)^{1/2})$.

In general to find the required irreducible factors of the jth such polynomial it is necessary to factor it over F_{j-1}. As was mentioned earlier, Wang and Weinberger have recently discussed algorithms for factoring over algebraic number fields and van der Waerden [vdW 49] presents an algorithm for factoring polynomials over an algebraic extension of the field $Q(x_1,\ldots,x_n)$. However, except for special cases, these algorithms are quite slow.

Fortunately in most cases the pure polynomials that arise can be proved irreducible a priori and in such cases no factoring has to be done at all. One case in which the pure polynomials can be reducible is when the degree t is even, B is an integer, and a root of unity has been adjoined to the field in which the coefficients of the factor must lie. For example $y^2 - 2 = [y - (\omega_8 - \omega_8^3)][y + (\omega_8 - \omega_8^3)]$ since $\omega_8 - \omega_8^3 = \sqrt{2}$ where ω_8 is the primitive eighth root of unity $e^{i\pi/4}$. In RADCAN, it was decided not to attempt such factorizations in order to make the algorithm faster. Hence, as was discussed in the previous section RADCAN does not handle expressions involving roots of unity and even roots of integers in a completely general way. The theorems on the irreducibility of the pure equations are presented in section 4.

Let us return to step (b) in the above procedure. In general, given P, there are many sets B that form a basis for P. For example if $P = \{24, x^4+6x^3+13x^2+12x+4, x^2+2x+1\}$ then $\{24,x^2+4x+4,x^2+2x+1\}$, $\{6,x+2,x+1\}$ and $\{2,3,x+2,x+1\}$ are all bases for P. The extension field found depends not only on the rules (i) - (iii) but the basis B also. In order for the irreducibility theorems to apply the elements of B must be square-free and pair-wise relatively prime. A basis in which each element is square-free is called a square-free basis. But even a square-free basis is not necessarily unique.

There are two natural ways to compute a square-free basis. One way is to let B consist of all the distinct irreducible factors of elements in P. A second way is to compute a set $\bar{B}$ of all square-free factors of elements in P. For polynomials of positive degree square-free factorization is much faster than factoring the polynomials into irreducible factors. See Yun [these Proceedings, pp. 248-259] for square-free factorization algorithms.

$\bar{B}$ may be refined into a square-free basis if for each pair B_1,B_2 in $\bar{B}$ with $\gcd(B_1,B_2) = C \neq 1$ we replace B_1 and B_2 by the non-units among (B_1/C), (B_2/C), and C. This process must be repeated until the elements of $\bar{B}$ are pair-wise relatively prime at which time we will have the desired basis B. All of these operations can be carried out fairly efficiently with the exception of computing square-free factors of integers. The only method known for computing square-free factors of integers is to completely factor them into primes. The best of the known algorithms for factoring an integer requires time that is exponential in the length of the integer.

With the form 1 option RADCAN computes a basis in which each element is irreducible. With the form 2 option RADCAN computes a basis in which the integers are prime and the polynomials of positive degree are square-free and pair-wise relatively prime.

This ends our discussion of the first aspect of the interpretation and simplification of radical expressions. We still need to give an intrepretation for the symbols that describe the algebraic extension fields found by the above procedure.

This has been done precisely and elegantly by R. H. Risch [Ris 69a] for univariate functions. For simplicity we also shall specify interpretations only for univariate functions. Interpretations for multivariate expressions can be specified in a similar way. See, for example [Eps 75, pp.161-163] for an interpretation of multivariate exponential and logarithmic expressions.

Suppose S is a list of symbols and let E(S) be the smallest set of expressions containing the elements of S with the property that whenever a,b are in S, so are a + b, a - b, a * b and a/b. For our purposes S is always of the form $S = (1, x, \gamma_1,\ldots,\gamma_m)$ where $m \geq 0$. For each γ_i there is a P_i in E((1,x)) and an r_i in E((1)) such that γ_i is the symbol $P_i^{r_i}$. Roughly speaking, E((1)) denotes rational numbers and $P_i^{r_i}$ denotes a polynomial raised to a rational power. E(S) is a special case of Risch's elementary field descriptions and shall be called an algebraic field description. An interpretation of E(S) is a pair (Mer(A),I) where Mer(A) is a field of single-valued algebraic functions which are meromorphic on an open, connected subset A of the complex plane. I is a mapping from a subset of E(S), namely that subset of expressions with a well-defined interpretation, onto Mer(A) such that

(i) I(1) is the constant function 1;
(ii) I(x) is the identity function on A;
(iii) For each γ_i symbol P^r, let s/t = I(r) where s and t are relatively prime integers with t > 0. Then $I(\gamma_i)^t - I(P)^s = 0$. If $I(\gamma_i)^t - I(P)^s$ is the monic irreducible equation satisfied

by $I(\gamma_j)$ over $Q(I(x), I(\gamma_1),\ldots,I(\gamma_{j-1}))$ then E(S) is regular; otherwise E(S) is non-regular.

(iv) If a and b are in the domain of I, so are a + b, a - b, and a * b. If $I(b) \neq 0$, a/b is also in the domain of I. Furthermore, $I(a \circ b) = I(a) \circ I(b)$ when $a \circ b$ is in the domain of I and $\circ$ is one of +, -, *, or /.

A regular algebraic field description E(S) has the desirable property that all interpretations of it are differentially isomorphic (see [Ris 69a], p.176, proposition 2.3 for a proof of this fact). This is the previously alluded to mathematical result that means for computational purposes all results of performing field operations and differentiation on the elements of E(S) are independent of the interpretation, i.e., independent of the single-valued branches of the multi-valued functions chosen.

Normally it is not necessary to distinguish the interpretation of a symbol from the symbol itself and we shall not do so henceforth.

Let us now consider the problem of substitution in a radical expression, i.e., we are given a radical expression f(x) and a polynomial P(x) and we wish to substitute P(x) for x in f(x) to obtain f(P(x)). When P(x) is a constant we have the evaluation problem.

We restrict ourselves to substituting polynomials for x. If radical expressions are substituted for x we can obtain nested radical expressions. Our restriction insures that f(P(x)) will be an unnested radical expression.

It is easy to see that one can obtain completely general unnested radical expressions through the process of substituting in a multivariate radical expression. Thus the substitution process is equivalent to the simplification process itself. Thus the interpretation of the symbol f(P(x)) can be given by our previous discussion and the "value" of f(P(x)) can be computed by applying our simplification process. In general when evaluating f(P(x)) one already has an algebraic extension F of Q(x) in which f(x) lies. It is only necessary to extend F and its corresponding basis to find an algebraic extension of F in which f(P(x)) lies. It is not necessary to repeat the entire process of finding an algebraic extension of Q(x) for f(P(x)).

With this view of substitution and our previous rules for resolving ambiguities and interpreting radical expressions, we have a generalization of Fateman's positive real interpretation [Fat 72] of radical expressions. The interpretation given herein effectively chooses the positive real branch for algebraic functions when the radical expression is ambiguous as it may be initially or as it may be after substitution. But once the ambiguities are solved and the extension field found, the field operations can be carried out in a manner that is independent of the interpretation. Thus there is no conflict between the regular field extensions of Risch and the positive real interpretations of Fateman that are used in MACSYMA.

4. Irreducibility Theorems for Pure Polynomials

In this section we present the irreducibility theorems for the pure polynomials discussed earlier. The first theorem tells us that $y^m - p$, p a prime integer is irreducible over any algebraic number extension of Q(x) obtained by adjoining roots of other prime integers. It is important that roots of unity not be contained in tne extension field for then the result is false as was shown by the example in which $y^2 - 2$ factors when $e^{1\pi/4}$ is in the extension field.

Theorem 1. Let m be a positive integer, $p_1,\ldots,p_k$ distinct positive prime integers. Let $p_i^{1/m}$ denote an mth root of p_i. Then the field $Q(p_1^{1/m},\ldots,p_k^{1/m})$ is of degree m^k over Q.

For a proof of this theorem see [Fat 72, pp. 135-140] or [Ric 74].

Corollary 2. Let $m_1,\ldots,m_k$ be positive integers; $p_1,\ldots,p_k$ be distinct positive prime integers; and let p_i^{1/m_i} denote an m_i^{th} root of p_i. Then $y^{m_k} - p_k$ is irreducible over $Q(p_1^{m_1},\ldots,p_{k-1}^{m_{k-1}})$.

Proof. Let $m = \ell cm(m_1,\ldots,m_k)$. By theorem 1 $y^m - p_k$ is irreducible over $G = Q(p_1^m,\ldots,p_{k-1}^m)$ which contains $F = Q(p_1^{m_1},\ldots,p_{k-1}^{m_{k-1}})$. Hence $y^m - p_k$ cannot factor over F without factoring over G. ∎

All examples known to us in which $y^m - p$ factors involve roots of unity with m even. It would be nice to know if $y^{m_k} - p_k$ factors over $Q(\omega_\ell, p_1^{1/m_1},\ldots,p_{k-1}^{1/m_{k-1}})$ where $p_1,\ldots,p_k$ are distinct primes, m_k is odd, and, ℓ, $m_1,\ldots,m_{k-1}$ are arbitrary positive integers. The next theorem provides a partial answer to this question.

Theorem 3. Let ℓ be a positive integer, m an odd positive integer, and $p_1,\ldots,p_k$ be distinct positive prime integers. Then the field $Q(\omega_\ell, p_1^{1/m},\ldots,p_k^{1/m})$ is of degree m^k over $Q(\omega_\ell)$ where ω_ℓ is a primitive ℓ^{th} root of unity.

For a proof see [Cav 68, pp.50-54]. The three results above still hold under the weaker hypotheses that the p_i are square-free and pairwise relatively prime.

If the hypotheses of theorem 3 are changed so that ℓ is odd and m is any arbitrary positive integer the result is false since $\sqrt{5} = -2\omega_5^2 - 2\omega_5 - 1$.

The final theorem tells us that in our algorithm $y^m - B(x_1,\ldots,x_n)$ is always irreducible where B is a non-constant polynomial in $Q(x_1,\ldots,x_n)$.

Theorem 4. Let m be a positive integer, $B_1,\ldots,B_k$ be non-constant, square-free, pair-wise relatively prime polynomials in $Q[x_1,\ldots,x_n]$. Then the field

$C(x_1,\ldots,x_n)(B_1^{1/m},\ldots,B_k^{1/m})$ is of degree m^k over $C(x_1,\ldots,x_n)$ where C denotes the field of complex numbers and $B_i^{1/m}$ denotes any one of the roots of $y^m - B_i = 0$.

The proof in [Cav 68, pp.54-59] can be modified in a straightforward way to give the above result. In [Fat 72, pp.135-140] a slightly weaker result is proved with C replaced by Q and with the added hypothesis that the B_i be positive or m be odd.

Theorem 4 has a natural corollary that is analogous to the corollary of theorem 1 and is proved in the same manner. Hence we will not actually state it.

5. Computing Times

In this section we present a brief discussion of the worst case computing time of some of the algorithms involved in the simplification process. The length of a non-zero integer n, denoted L(n), is $\lfloor \ell og_2|n| \rfloor + 1$. $L(0) = 1$. Let $Z[n,d,a]$ denote the set of polynomials in $Z[x_1,\ldots,x_n]$ with degree in each variable $\leq d$ and the length of each integer coefficient $\leq a$.

One of the major procedures in the simplification process is the computation of a basis $\mathcal{B}$ for $\mathcal{P}$, the set of radicands. Suppose each element of $\mathcal{P}$ is in $Z[n,d,a]$.

In [Eps 76] Epstein presents carefully analyzed algorithms for computing a square-free basis for polynomials with Gaussian integer coefficients. We assume straightforward changes to Epstein's algorithms and computing time analyses to the restricted case that we are considering, namely polynomials with rational integer coefficients.

We have also defined basis in a slightly different manner from Epstein - his definition ignores constants. Thus for our purposes we can compute a square-free basis by first finding the integer content and primitive parts of each of the non-constant polynomials in $\mathcal{P}$. Let $\mathcal{P}_1$ = the set obtained by the union of the constants in $\mathcal{P}$ and the integer contents of the non-constant members of $\mathcal{P}$. Let $\mathcal{P}_2$ be the set of primitive parts of non-constant members of $\mathcal{P}$. Let $\mathcal{B}_1$ be a basis for $\mathcal{P}_1$ and $\mathcal{P}_2$ be a basis for $\mathcal{B}_2$. Then $\mathcal{B}_1 \cup \mathcal{B}_2$ will be our desired basis for $\mathcal{P}$. Assume the elements of $\mathcal{P}_1$ and $\mathcal{P}_2$ are in $Z[n,d,a]$ and that $\mathcal{P}_1$ contains k_1 distinct integers while $\mathcal{P}_2$ contains k_2 distinct polynomials. Then $\mathcal{B}_1$ can be computed by factoring each element into prime factors which can be done in time bounded by a function which is $O(k_1 \cdot 2^{a/2})$. The best known algorithms for factoring integers have somewhat better computing time bounds but they still require time that is an exponential function of a. Hence we will be content with the $O(2^{a/2})$ bound.

Epstein's algorithms will compute $\mathcal{B}_2$ in time bounded by a function which is $O(k_2^3 a^2 d^{3/2}(d+1)^{n+1})$. Furthermore Epstein's algorithm GPPSQF (Gaussian Polynomial Primitive Square-Free Factorization) can be trivially modified to provide the desired integer contents of the non-constant elements of $\mathcal{P}$ without additional work. His algorithm as currently constituted actually computes the integer contents but ignores it. Thus the entire basis computation can be done in time bounded by a function which is $O(k^{1/2a} + k^3 a^2 d^{3/2}(d+1)^{n+1})$ where $k = \max(k_1,k_2)$.

This computing time bound was obtained assuming that the modular greatest common divisor algorithm is used for gcd computations and that the assumptions made by Brown [Bro 71] in analyzing the modular gcd algorithm hold.

In RADCAN's phase II rational simplification is performed on rational functions in $n+\nu$ variables where ν is the number of elements in $\mathcal{B}$.

A set of integers, all bounded in magnitude by A, has at most $\ell og_2 A$ distinct prime factors. Thus $\mathcal{B}$ contains at most $a+1$ integers. If each polynomial in $\mathcal{P}$ should factor into distinct linear factors in $\mathcal{B}$ then $\mathcal{B}$ would contain at most $k_2 nd$ non-constant polynomials.

Thus the rational functions in phase II of RADCAN have at most $k_2 nd + a + n + 1$ variables. Hence the time for the gcd calculation in this phase, assuming the modular algorithm is used, is $O((d+1)^{k_2 nd+a+n+1})$ assuming each of the polynomials has degree at most d in each of the variables.

Hence one can see that the bounds for the worst case computing times for phases I and II of RADCAN grow very rapidly as a, n, d and k_2 increase.

Although these algorithms can require a lot of computing time in practice, experience with MACSYMA indicates that RADCAN is a reasonable algorithm to use for routine simplification of small expressions. In fact it was, at one time, a default preliminary step in simplifying single algebraic equations prior to their solution by the SOLVE command [Mar 71].

6. An Alternative Approach and a Review of Related Research

As is well-known each multiple algebraic extension field of Q(x) can be expressed as a single extension. This statement is known as the theorem on the primitive element [vdW 49]. From van der Waerden's discussion algorithms can be constructed to find an element β such that $Q(x)(B)$ is isomorphic to $Q(x)(\alpha_1,\ldots,\alpha_n)$ where $\alpha_1,\ldots,\alpha_n$ are algebraic over Q(x). Loos[Loo 73] also discusses constructive aspects of the theorem on the primitive element in relationship to performing arithmetic in the field of all algebraic numbers.

So since the theorem on the primitive element can be constructively implemented, why use multiple algebraic extensions when one would do. There seem to be at least two reasons for not simplifying radical expressions via the approach suggested by the theorem on the primitive element. First it seems likely, although there has been no definitive study to verify it, that multiple algebraic extensions of lower degree lead to faster algorithms than would one extension of high degree. The degree of the one extension would be the product of

the degrees of the multiple extensions.

Secondly multiple algebraic extensions seem to possess better human engineering attributes than extensions obtained through the theorem on the primitive element. For example the field $Q(\sqrt{3}, \sqrt{5})$ is contained in the field $Q(a)$ where a satisfies the irreducible polynomial $M(a) = a^4 - 16a^2 + 4$. In $Q(a)$, $\sqrt{3}$ is written as $1/4(a^3 - 14a)$ which from a human engineering point of view is generally less desirable than simply $\sqrt{3}$.

So for these reasons replacing multiple algebraic extensions by a single extension seems undesirable.

There have been a number of other papers written that are related to our discussion in one fashion or another. We will give a brief discussion of the related work. Much of it has already been mentioned in the preceding sections. First it should be noted that the present paper is based on the authors' Ph.D. dissertations [Cav 68] and [Fat 72].

Many papers have been written on various aspects of algebraic simplication. We will discuss only those that treat algebraic numbers and algebraic functions. For an overview of other aspects of simplification up to 1971, see the paper by Moses [Mos 71]. Further much of the material in classical modern algebra on algebraic extension fields is relevant to this work. van der Waerden [vdW 49] contains a treatment of this material witn many algorithmic methods given.

The first significant paper to deal explicitly with simplification of algebraic functions was an unpublished Bell Laboratories report by S. L. Kleiman [Kle 66]. Kleiman considers some problems that are closely related to the problems considered here. He assumes that he is given a set of irreducible algebraic equations that define the algebraic dependencies among the variables involved. Thus he avoids discussing the problem of transforming a set of radical expressions into a set of pure polynomials and the factoring of such polynomials over algebraic extensions of Q. However, his paper is precisely written, contains interesting ideas, and should be more widely available and better known. For example, if the polynomials generate a prime ideal, Kleiman's results constructively refute Loos' [Loo 74] conjecture that given a set of polynomial equivalences, there does not exist a canonical form for a polynomial under these relations.

Fitch [Fit 71] discusses the general problem of algebraic simplification. From his work in general relativity in which unnested radical expressions occurred extensively he was lead to consider in some detail the problem of simplifying radical expressions. His solution has much in common with the methods described in this paper although he neither discusses interpretations of radical expressions nor deals with the problems associated with roots of unity.

The more accessible [Fit 73] covers much of the same ground as [Fit 71]. In addition it contains a simple proof, which he attributed to M. N. Huxley, of the fact that $(1+x)^{1+x}$ is transcendental over $Q(x)$. This result also follows from the more general structure theorem of Risch [Ris 69b].

Rubald [Rub 73] developed algorithms for polynomial arithmetic over real algebraic number fields $Q(a)$ where the extension is simple. His algorithms do not require an irreducible defining polynomial for a. He also presents an algorithm for determining if $\alpha < \beta$ where α and β are real algebraic numbers. Such a determination is dependent on the particular root of the defining equation joined to Q and could not be realized by the methods discussed here.

Recently Shtokhamer has written on matters related to the work of Kleiman [Sht 75a] and on the simplification of nested radicals [Sht 75b]. In [Sht 75a] he claims to have developed algorithms for finding unique representatives of equivalence classes of $R[x_1,\ldots,x_n]$ modulo an ideal where R is a Noetherian domain. In [Sht 75b] he generalizes the methods given in the authors' dissertations to apply to nested radical expressions. His algorithms are programmed in REDUCE.

M. Lauer [Lau 76] brings together the algorithms of Shtokhamer [Sht 75a] and an earlier algorithm of Buchberger [Buc 70] to solve the problem of finding canonical representatives for the equivalence classes of $S[x_1,\ldots,x_n]$ modulo an ideal when S is either a field or a principal ideal domain.

The recent paper by Wang [Wan 75] reports on the progress of a program to factor polynomials over algebraic number fields. While efficiencies may be realized in special cases, the general computational intractability has not been resolved.

Although the irreducibility theorems of section 4 may have been known to some members of the mathematical community for many years, the first published results appeared in 1940 [Bes 40] when Besicovitch proved Theorem 1. In 1972 [Fat 72] Fateman gave an independent proof of theorem 1. In 1968 Caviness [Cav 68] presented theorems 3 and 4, and in 1974 Richards [Ric 74] published theorem 1 and theorem 3 for even ℓ.

Acknowledgments

The authors wish to thank their colleagues at Berkeley, MIT, the University of Wisconsin, and Bell Telephone Laboratories for stimulating discussions. This work was supported in part by Project MAC, an MIT interdepartmental laboratory sponsored by the Advanced Research Projects Agency (ARPA), Department of Defense, under Office of Naval Research Contract N00014-70-A-0362-0001. One author (RJF) was supported, while a graduate student, by ARPA under Air Force contract F19628-68-0101 with Harvard University, by the National Science Foundation under their Graduate Traineeship program and by Bell Telephone Laboratories under a contract with Harvard University. The other author (BFC) was supported, while a graduate student, by the National Science Foundation, in the form of a graduate fellowship. Professors Alan Perlis of Yale University and

Henry Leonard of Northern Illinois University made valued contributions to this work. The comments of John Fitch, Peter Weinberger, and the referees about an earlier version of the paper were also helpful.

References

[Bes 40] A. S. Besicovitch, On the Linear Independence of Fractional Powers of Integers, J. London Math. Soc. 15 (1940), 3-6.

[Bro 71] W. S. Brown, On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors, J. ACM 18, 4 (Oct. 1971), 478-504.

[Buc 70] B. Buchberger, Ein Algorithmisches Kriterium für die Lösbarkeit eines Algebraischen Gleichungssystems, Aequationes Mathematicae 4 (1970).

[Cav 68] B. F. Caviness, On Canonical Forms and Simplification, Ph.D. Dissertation, Carnegie-Mellon University (May, 1968), 80 pages. Available from Xerox University Microfilms, Ann Arbor, Michigan.

[Cav 70] _____, On Canonical Forms and Simplification, J. ACM 17 2 (April 1970), 385-396.

[Col 74] G. E. Collins, Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition - Preliminary Report, SIGSAM Bulletin 8, 3 (August 1974), 80-90.

[Eps 75] H. I. Epstein, Algorithms for Elementary Transcendental Function Arithmetic, Ph.D. Dissertation, University of Wisconsin, (May 1975), 408 pages. Available from Xerox University Microfilms, Ann Arbor, Michigan.

[Eps 76] _____, Using Basis Computation to Determine Pseudo-Multiplicative Independence, these Proceedings.

[Fat 72] R. J. Fateman, Essays in Algebraic Simplification, Ph.D. Dissertation, Harvard University. Revised version reprinted as MIT Project MAC Tech. Report MAC TR-95 (April 1972), 190 pages.

[Fit 71] John P. Fitch, An Algebraic Manipulator, Ph.D. Dissertation, University of Cambridge (Oct. 1971).

[Fit 73] _____, On Algebraic Simplification, Computer J. 16 (1973), 23-27.

[GJY 75] J. H. Griesmer, R. D. Jenks, and D. Y. Y. Yun, SCRATCHPAD Users Manual, Report RA 70, IBM Research Center, Yorktown Heights, N.Y. (June 1975), 66 pages.

[Kle 66] S. L. Kleiman, Computing with Rational Expressions in Several Algebraically Dependent Variables, Bell Laboratories Tech. Report, Murray Hill, New Jersey, (1966), 40 pages. Reprinted as Computing Science Tech. Report #42 (1976).

[Knu 69] D. E. Knuth, The Art of Computer Programming, vol. 2 "Semi-numerical Algorithms," Addison-Wesley (1969).

[Lau 76] Markus Lauer, Canonical Representatives for Residue Classes of a Polynomial Ideal, these Proceedings.

[Loo 73] R. Loos, A Constructive Approach to Algebraic Numbers, preprint.

[Loo 74] _____, Toward a Formal Implementation of Computer Algebra, SIGSAM Bulletin 9, 3 (August 1975), 21-23.

[MaF 71] W. A. Martin and R. J. Fateman, The MACSYMA System, Proceedings of Second Symposium on Symbolic and Algebraic Manipulation (March 1971).

[Mat 75] Mathlab Group, MACSYMA Reference Manual, The Laboratory for Computer Science, M.I.T., Cambridge, Massachusetts, (November 1975), 199 pages.

[Mos 71] Joel Moses, Algebraic Simplification: A Guide for the Perplexed, Comm. ACM 14, 8 (August 1971), 527-537.

[Pol 50] Harry Pollard, The Theory of Algebraic Numbers, The Mathematical Association of America (1950).

[Ric 74] Ian Richards, An Application of Galois Theory to Elementary Arithmetic, Adv. in Math. 13 (1974), 268-273.

[Ris 69a] R. H. Risch, The Problem of Integration in Finite Terms, Trans. AMS, 139 (May 1969), 167-189.

[Ris 69b] _____, Further Results on Elementary Functions, IBM Tech. Report RC 2402, Yorktown Heights, N.Y. (March 1969).

[Rub 73] C. M. Rubald, Algorithms for Polynomials Over a Real Algebraic Number Field, Ph.D. Dissertation, University of Wisconsin (1973), 224 pages.

[Sht 75a] Roman Shtokhamer, Simple Ideal Theory: Some Applications to Algebraic Simplification, University of Utah Tech. Report UCP-36 (July 1975), 22 pages.

[Sht 75b] _____, Simplification of Nested Radicals, University of Utah Tech. Report UCP-37 (July 1975), 16 pages.

[vdW 49] B. L. van der Waerden, Modern Algebra, tr. F. Blum, Frederick Ungar Publ. Co. (1949).

[Wan 75] Paul Wang, Factoring Multivariate Polynomials Over Algebraic Number Fields in MACSYMA, SIGSAM Bulletin 9, 3 (August 1975), 21-23.

[Wei 76] Peter Weinberger, Factoring Polynomials Over Algebraic Number Fields, ACM Trans. on Math. Software (to appear).

Canonical Representatives for Residue Classes of a Polynomial Ideal

Markus Lauer

Universität Kaiserslautern

March 1976

Abstract

At the EUROSAM Conference 1974 it was conjectured that there exist probably no canonical forms for polynomial expressions under polynomial side relations. In 1965, however,Buchberger had already implemented a reduction algorithm solving the problem over fields. Positive results were also attained independently since 1974 by R. Shtokhamer and the author. In this note a general theorem on canonical representatives is obtained and by application to Buchberger's and Shtokhamer's algorithms it is proven that the stated problem is solved .

1. Introduction

In Computer Algebra systems operations are performed on algebraic objects having an extended structure. Usually many equivalent structures represent the same object. However,the algorithms require the collection of equivalent terms, the dropping of terms equivalent to 0 and similar simplifications. It is therefore advantageous for algebraic algorithms to work on unique representatives of equivalent structures, called canonical forms of the algebraic objects. Canonical forms for large classes of expressions were studied by Caviness [1], Brown [2], Moses [3] and Kleiman [15].

Frequently in practice algebraic operations have to be performed in the context of side relations, user defined simplification rules, pattern matching or similar machinery- invented for the convenience of the user. Naturally the questions arise [4], is there an algebraic meaning of such manipulations, can canonical forms still be defined in such an environment? The practical goal is of course to utilize the side relations to the utmost point for the reduction of expressions. For the uninitiated reader it may be surprising that the semantics of an expression evaluator using a rule stack [SCRATCHPAD,5] is intimately related to chain conditions and ideal theory .

2. Statement of the Problem

Before we give an example we start with a precise statement of the mathematical problem :

Let S be a field (case 1) or a principal ideal domain (case 2) , let $R = S[x_1,\ldots,x_r]$ $(r \geq 1)$ and $U = (f_1,\ldots,f_s)$ the ideal generated by $f_1,\ldots,f_s$, members of R . Find a computable function $F: R \to R$ with properties

(1) $F(g) - g \in U$, for all $g \in R$
(2) $F(g) = F(h) \Leftrightarrow g - h \in U$, for all $g,h \in R$.

Also F(g) should be "simpler" than g in a sense which has to be made precise .

As example consider the two side relations as they may be realized by LET statements

$$3x^2y = 3x^2 - y^2 + 2 \;;$$
$$4xy = -1 \;;$$

First we note that x and y may stand for more general expressions like kernels , so polynomial expressions are not as restrictive in practice as it may appear at a first glance . Intuitively the meaning of the side relations is the replacement of the two polynomials

$$f_1 = 3x^2y - 3x^2 + y^2 - 2 ,$$
$$f_2 = 4xy + 1$$

by 0 "as often as possible" in all following expressions . (By this interpretation there is no restriction to monomials on the left hand side of the side relations.) The hard problem for an algebraic system is to draw all conclusions from the side relations because otherwise large expressions equivalent to 0 can occur, causing space to be exhausted. In the example, there are polynomials A and B $\in$ R such that the expression (in y only!)

$$g = 16y^4 - 32y^2 + 3y - 3 = Af_1 + Bf_2 = 0$$

vanishes - a hard test for a symbolic system . Now, the set of all polynomials of the form $Af_1 + Bf_2$ for arbitrary A,B $\in$ R is just the ideal U generated by f_1 and f_2 and the problem studied for a given g is how can g be reduced with respect to U.

For both cases of the coefficient domain S the problem is solved. If S is a field (case 1) the solution is a by-product of a result of Buchberger [6] giving an algo-

rithmic criterion for the solvability of a system of algebraic equations. If S is a principal ideal domain (case 2) the solution is obtained by generalizing the notion of a Szekeres basis [7,9] to Noetherian rings .

In both cases the main work consists in constructing for the ideal a suitable basis which in a natural way allows the definition of the desired function F. In case 1 this is accomplished by Buchberger's algorithm, in case 2 by an algorithm due to Shtokhamer [7] .

After establishing a general theorem on canonical representatives in the next section, it is applied in the remaining two sections to both algorithms, allowing a unified treatment of these two independent approaches .

3. Canonical Representatives

The most extensive generalization of the problem is to treat it in terms of equivalence relations as will be done in this section .

3.1 Definition : Let M be a set, ~ an equivalence relation on M. An ample function [10,11] (af) for (M,~) is a mapping $f : M \to M$ such that :

(A1) $f(a) \sim a$ for all $a \in M$
(A2) $a \sim b \Rightarrow f(a) = f(b)$ for all $a,b \in M$.

Two evident properties of an af are stated in the following lemma .

3.2 Lemma : Let f be an af for (M,~).Then:

(1) $f^2 = f$
(2) $f(a) = f(b) \Rightarrow a \sim b$ for all $a,b \in M$.

Proof: (1) (A1): $f(a) \sim a \Rightarrow$ (A2) $f^2(a) = f(a)$.

(2) let $f(a) = f(b)$. Then:

$a \sim f(a) = f(b) \sim b$. ∎

Now it is clear what a canonical representative is :

3.3 Definition : Let f be an af for (M,~). For $a \in M$ let $\tilde{a}$ denote the equivalence class containing a . The unique element $a^* \in \tilde{a}$ for which $f(a^*) = a^*$ holds is called a canonical representative for $\tilde{a}$.

Of course there are usually several possibilities to define canonical representatives for the same equivalence relation . For example set M = Z (the integers) and $a \sim b$ iff $a \equiv b \bmod n$ for some fixed positive $n \in Z$. For $a \in Z$ there exist unique $r_1, r_2 \in Z$ such that :

$$a = q_1 n + r_1 \ , \ 0 \le r_1 < n$$

and $$a = q_2 n + r_2 \ , \ -\frac{n}{2} < r_2 \le \frac{n}{2}$$

Then f_1 and f_2 defined by $f_i(a) = r_i$, i=1,2, are both af's. One way to select a special af is to define a suitable ordering on M .

3.4 Definition : Let f be an af for (M,~). Let $\le$ be a partial ordering for M. Then f is called a simplifying ample function (saf) for $(M,\sim,\le)$ if:

(A3) $f(a) \le a$ for all $a \in M$.

As an immediate consequence of (A2) we have :

3.5 Lemma : Let f be a saf for $(M,\sim,\le)$.

Then $f(a) \le b$ for all $b \in \tilde{a}$.

That condition (A3) is indeed sufficient for choosing a unique af is the content of the next theorem .

3.6 Theorem : There exists only one saf for $(M,\sim,\le)$.

Proof: Let f_1 and f_2 be two saf's for $(M,\sim,\le)$. Clearly $f_1(a) \sim f_2(a)$ for all $a \in M$. Therefore by lemmas (3.2) and (3.5):

$$f_1(a) = f_1^2(a) \le f_2(a)$$

and analoguesly $f_2(a) \le f_1(a)$. ∎

It is clear that a saf is unique only with respect to the chosen ordering. In the example above with M = Z mod n f_1 is a saf for the following ordering $\le_1$:

$$a <_1 b \iff \begin{cases} a = 0 \ \& \ b \neq 0 \\ a > 0 \ \& \ b < 0 \\ \mathrm{sign}(a) = \mathrm{sign}(b) \ \& \\ |a| < |b| \end{cases}$$

and $a \le_1 b \iff a = b$ or $a <_1 b$.

Similarly, f_2 is a saf for the ordering $\le_2$:

$$a <_2 b \iff \begin{cases} |a| = |b| \ \& \ a > b \\ |a| \neq |b| \ \& \ |a| < |b| \end{cases}$$

and $a \le_2 b \iff a = b$ or $a <_2 b$.

It is easy to see that there exists no saf for Z with the normal ordering $\le$ and the congruence relation.

As a converse of lemma(3.5) we have the following existence lemma for saf's :

3.7 Lemma : If each equivalence class $\tilde{a}$ of $(M,\sim,\le)$ contains a minimal element a^*, then the mapping

$$f: b \to a^* \text{ for all } b \in \tilde{a}$$

is a saf .

Proof: obvious .

For practical purposes a saf should also be a computable (recursive) function, an aspect which has been neglected until now .In [8], B. Caviness et al. give an existence lemma for computable af's.

A computable af is called canonical form. If M is an additive group then condition (A2) is weakened to:

(A2') $a \sim b \sim 0 \Rightarrow f(a) = f(b)$ for all $a,b \in M$.

A computable mapping $f: M \to M$ with (A1) and (A2') is called a normal form. The existence lemma is :

Lemma : (Caviness et al.)

If M is recursively enumerable and if there

is a normal form for $(M,\sim)$, then there is a canonical form for $(M,\sim)$.

The canonical form they construct is the same as in lemma(3.7). They use the fact that a recursively enumerable set is well-ordered by it's Gödel numbering, and then use the computability of the normal form and the Gödel numbers to show that the function f of lemma(3.7) is computable.

For the application to the problem of section 2 the set M is the polynomial ring R in several indeterminates and the equivalence relation is the congruence modulo an ideal U ($g \sim h \iff g - h \in U$). If the coefficient domain is recursively enumerable, then R is recursively enumerable; however the problem of finding a normal form for R seems to be not easier than finding a canonical form. We shall use the algebraic properties of R and U for finding a computable saf for $(M,\sim)$ with a suitable order $\leq$.

4. The Case of a Field (Buchberger's Algorithm)

Let S be a field, $R = S[x_1,\ldots,x_r]$, and $U = (f_1,\ldots,f_s)$ be an ideal in R. The equivalence relation $\sim$ is defined as

$f \sim g \iff f - g \in U$ for all $f,g \in R$.

We have

$$f - g \in U \iff f + U = g + U \in R/U.$$

If we have a linearly independent basis B for the vector space R/U (over S) then each residue class $g + U$ has a unique representation

$$g + U = g_1 b_1 + U + \ldots + g_k b_k + U,$$

where $g_i \in S$ and $b_i + U \in B$, $i = 1,\ldots,k$. Hence the polynomial $g_1 b_1 + \ldots + g_k b_k \in R$ can be considered as a canonical representative for $\tilde{g}$, and an algorithm which finds for each $g \in R$ the representation of $g + U$ in the basis B can be considered as a computable ample function. Therefore the goal is to find a linearly independent basis B for R/U and such an algorithm. The following solution is taken from Buchberger [6] where the omitted proofs can be found.

S being a field we may assume that the $f_1,\ldots,f_s$ of the ideal basis are monic.

$$f_j = x_1^{I_{j,1}} \ldots x_r^{I_{j,r}} + \sum a^{(j)}_{i_1\ldots i_r} x_1^{i_1} \ldots x_r^{i_r} = \mathrm{lt}(f_j) + \mathrm{red}(f_j) \tag{1}$$

where $\mathrm{lt}(f_j)$ denotes the leading term and $\mathrm{red}(f_j)$ the reductum of f_j. The terms are ordered with respect to the total degrees and lexicographically otherwise. Clearly, the set of all residue classes of all monomials $x_1^{i_1} \ldots x_r^{i_r}$ is a basis for R/U which generally is linearly dependent unless $U = (0)$. To construct a linearly independent subset Buchberger observes for an arbitrary term $x_1^{i_1} \ldots x_r^{i_r}$

$$x_1^{i_1} \ldots x_r^{i_r}\, \mathrm{lt}(f_j) \sim -\, x_1^{i_1} \ldots x_r^{i_r}\, \mathrm{red}(f_j) \tag{2}$$

since $f_j \in U$. The left hand side is a multiple of $\mathrm{lt}(f_j)$. A monomial which is not a multiple of any $\mathrm{lt}(f_j)$ is called a kernel and one defines

4.3 Definition : An M-reduction of a polynomial $g \in R$ is the iterative replacement of all terms of g which are multiples of a $\mathrm{lt}(f_j)$ by the corresponding right hand sides of (2) as often as possible.

This definition differs slightly from the original one given in [6]; but all results stated below are valid for this definition, too (see [16]).

By an M-reduction multiple terms of the $\mathrm{lt}(f_j)$'s are eliminated by subtracting suitable multiples of the basis polynomials f_j. The process generalizes division - g is "divided" by the whole basis - and the result is the "remainder" of this "division".

The result is in general not unique but is a polynomial $\sim g$ with kernel terms only. Therefore the residue classes of all kernels with respect to $f_1,\ldots,f_s$ are a basis for R/U and the criterion holds :

4.4 Lemma : If the basis $\{f_1,\ldots,f_s\}$ of an ideal U in R has the property that all possible M-reductions of a polynomial in R give the same result, then the residue classes of the kernels with respect to $f_1,\ldots,f_s$ are a linearly independent basis for R/U .

In [16], Buchberger attributes the idea for this lemma to W. Gröbner. A basis which satisfies the condition of this lemma is therefore called a Gröbner-basis (abbreviated G-basis) .

Buchberger gives the following algorithm based on this lemma.

Let f_j and f_k be two different elements of the ideal basis, $\mathrm{lt}(f_j)$ and $\mathrm{lt}(f_k)$ their leading terms, respectively. Let $pp_{j,k}$ be the least common multiple of the leading terms, i.e.

$$pp_{j,k} = x_1^{L_{j,k,1}} \ldots x_r^{L_{j,k,r}}, \text{ where}$$

$$L_{j,k,t} = \max(I_{j,t}, I_{k,t}) \quad \text{(see eq.(1))}.$$

From equation (2) follows :

$$pp_{j,k} \sim -\sum a^{(j)}_{i_1,\ldots,i_r} x_1^{i_1+d_{j,1}} \ldots x_r^{i_r+d_{j,r}} \tag{3}$$

and

$$pp_{j,k} \sim -\sum a^{(k)}_{i_1,\ldots,i_r} x_1^{i_1+d_{k,1}} \ldots x_r^{i_r+d_{k,r}} \tag{4}$$

where $d_{j,t} = L_{j,k,t} - I_{j,t}$ and

$d_{k,t} = L_{j,k,t} - I_{k,t}$, $t = 1,\ldots,m$.

The polynomial s_{jk} obtained by subtraction of the right hand sides of (3) and (4) is called the S-polynomial for $pp_{j,k}$. Note that $s_{jk} = f_j\bar{a} - f_k\bar{b} \in R$ with $\bar{a} = pp_{j,k} / lt(f_j)$, $\bar{b} = pp_{j,k} / lt(f_k)$. The algorithm performs then an M-reduction of s_{jk} with respect to the present ideal basis. If the result is non-zero, it is made monic and joined to the ideal basis.

These operations are done for all pairs of different elements of the ideal basis, and also for those polynomials which are added to the basis.
Two special cases are regarded:

S1. $pp_{j,k} = lt(f_j)$ (or $pp_{j,k} = lt(f_k)$). Then, after the operations described are performed, the polynomial f_j (or f_k respectively) is removed from the basis.

S2. $pp_{j,k} = lt(f_j)*lt(f_k)$. Then the operations are not performed, and the next $pp_{j,k}$ is processed.

B ← GBASIS(A)

[Gröbner basis. $A = \{f_1,\ldots,f_s\}$, a basis for an ideal U in $S[x_1,\ldots,x_r]$. B is a G-basis for the same ideal U.]

(1) [Initial basis] B ← A.

(2) [Least common multiples] for all different pairs f_j , $f_k \in A$ include $c_{j,k} = lcm(lt(f_j),lt(f_k))$ in a list L unless $c_{jk} = lt(f_j)*lt(f_k)$.

(3) [Empty L] for each element c_{jk} of L perform step 4-7 and stop.

(4) [Subtraction] Let $c_{jk} = lcm(lt(f_j)$, $lt(f_k))$, set s_{jk} to the corresponding S-polynomial.

(5) [Reduction] M-reduce s_{jk} to $\bar{s}_{jk}$ over B. If $\bar{s}_{jk} = 0$, perform with the next element c_{jk} in L, if any, step 4-7.

(6) [Augment L] Join the monic associate of $\bar{s}_{jk}$ to B. Augment L by all $c_{si} = lcm(lt(f_i),lt(\bar{s}_{jk}))$ for all f_i in B unless $c_{si} = lt(f_i)*lt(\bar{s}_{jk})$.

(7) [Prune L] If $c_{jk} = lt(f_j)$ then remove f_j from B and remove all c_{je} and c_{qj} from L; similarly, if $c_{jk} = lt(f_k)$. ∎

By the M-reduction in step(5) the polynomials joined to the basis in step(6) contain only kernels with respect to the current basis B. Buchberger proves that a sequence $lt(f_1),lt(f_2),\ldots$ with the property that each $lt(f_k)$ is a kernel with respect to all $lt(f_m)$, $m < k$, has only a finite number of elements. This property generalizes an essential aspect of a degree function in a division algorithm and enforces the termination of the algorithm. He also shows that the basis produced has the property required in Lemma(4.4).

In order to apply our results of section 3 we let, for $g \in R$, $\bar{g} = F(g)$ denote the result of an M-reduction with respect to a G-basis and we extend the total ordering of terms of a polynomial to a partial ordering over R.

4.6 Definition : For $f,h \in R$ define $<$ by

$$g < h \iff \begin{cases} lt(g) < lt(h), \\ lt(g) = lt(h) \ \& \\ red(g) < red(h) \end{cases}$$

and define $g \leq h$ by $g = h$ or $g < h$.

For polynomials composed of the same monomials but differing in the numerical coefficients the order is undefined.

4.5 Theorem : An M-reduction with respect to a Gröbner basis is a computable simplifying ample function and therefore unique for all Gröbner bases generating the same ideal.

Proof: Clearly $F(g) \sim g$. Further $F(g)$ is a polynomial containing only kernels. The residue classes of kernels with respect to a G-basis are linearly independent by Lemma (4.4) which proves (A2). F is obviously computable. Since during an M-reduction a term is always replaced by smaller terms and since addition does not produce greater terms we have $F(g) \leq g$ proving (A3). Finally, it follows from Theorem(3.6) that an M-reduction over a G-basis is independent of the special G-basis. ∎

Starting with different bases generating the same ideal the algorithm GBASIS produces in general different G-bases $\{f_1,\ldots,f_s\}$ and $\{g_1,\ldots,g_t\}$. However, $s = t$, $lt(f_i) = lt(g_i)$, after a suitable reordering of the g_j's, which, too, proves the independence of an M-reduction over a G-basis.

The author conjectures that one obtains a unique G-basis if each element of a given G-basis is M-reduced with respect to the other elements. (This question has recently received a positive answer; see [16].)

Buchberger's algorithm applies in a remarkable way to the following special cases:

a) if the polynomials in the given ideal bases are linear then it is the usual Gaussian elimination algorithm for a system of linear equations.

b) for the univariate case ($r = 1$) it computes simultaneously the g.c.d. of the basis polynomials, i.e. it is a generalized Euclidean algorithm.

These observations may give a clue to an analysis of GBASIS.

5. The Case of a Principal Ideal Domain (Shtokhamer's Algorithm)

In the following we write pr.i.d. for a principal ideal domain which is an integral domain, too.

First we make the following generalization: let S be a Noetherian integral domain, i.e. an integral domain in which each ideal posesses a finite basis, and let $R = S[x]$. Let us further assume that there exist saf's for the congruence relations modulo each given ideal of S (given means that it is given by its basis). We will define a saf for each given ideal U of R and show its computability if the saf's for S are computable. As in the previous section we shall need a special kind of basis for U, namely

a generalized Szekeres basis. In [9], Szekeres constructed a canonical basis for the ideals of a polynomial domain in one indeterminate over a pr.i.d.
Let us generalize his construction (as far as necessary for our purposes) to Noetherian rings.

5.1 Definition : Let $\vec{G} = (g_1,\dots,g_n) \in R^n$ denote an n-tuple. If $\vec{A} = (a_1,\dots,a_n)$, $\vec{B} = (b_1,\dots,b_n) \in R^n$ then

$$\vec{A}\vec{B} := \sum_{i=1}^{n} a_i b_i$$

Then if $M = \{m_1,\dots,m_n\}$ is a basis of an ideal U in R , then

$$U = \{\vec{V}\vec{M} \mid \vec{V} \in R^n\}.$$

5.2 Definition : Let U be an idael of R. Let k be the least positive integer such that U contains a polynomial of degree k.

$L_i(U) := \{0\} \cup$ the set of leading coefficients of all polynomials in U of degree i, $i \geq k$. We shall omit the "U" if no confusion is possible.

5.3 Lemma : 1. L_i is an ideal in S for each integer $i \leq k$.
2. $L_i \subset L_{i+1}$ for all $i \geq k$.

Proof: trivial.

Let $A_j = (a_{j1},\dots,a_{jl_j})$ be a basis for L_j; since $A_j \subset L_j$, there exists for each $a_{ji} \in A_j$ a polynomial $p_{ji} \in R$ such that $\deg(p_{ji}) = j$ and the leading coefficient of p_{ji} is a_{ji}; let $G_j = \{p_{j1},\dots,p_{jl_j}\}$.

5.4 Definition : A finite set $G_j \subset U$ of polynomials of degree j with the property that the leading coefficients of the elements of G_j form a basis for L_j is called a minimal set (of U).

5.5 Lemma : Let G_k , G_{k+1} ,... be minimal sets in U;

let $M_n := \bigcup_{i=k}^{n} G_i$.

Then each $g \in U$ with $\deg(g) = s \leq n$ has a representation

$$g = \sum_{i=k}^{s} \vec{C}_i \cdot \vec{G}_i \text{ , where } \vec{C}_i \in S^{l_i}, \text{ and } l_i$$

is the number of elements of G_i.

Proof: obvious since the leading coefficients of the elements of G_i form a basis for L_i. ∎
An immediate consequence of Hilbert's basis theorem is :

5.6 Lemma : There is a $m \geq 0$ such that M_m is a basis for U but M_{m-1} is not a basis.
Of course the minimal sets are not uniquely determined by U. There exists a uniqueness condition for minimal sets, but for our intentions this is not necessary.

5.7 Definition : A basis M for an ideal U of R is called a SZ-basis if $M = \bigcup_{i=k}^{m} G_i$, where the G_i are minimal sets for U and k is the minimal degree of the polynomials in U.
From lemmas (5.3) and (5.5) follow the two properties of a SZ-basis :

5.8 Lemma : Let $M = \bigcup_{i=k}^{m} G_i$ be an SZ-basis for U; let $G_i = \{g_{i1},\dots,g_{il_i}\}$, A_i the set of leading coefficients of the elements of G_i. Then :

(1) for each $g_{ij} \in G_i$, $k \leq i < m$, there exist vectors $\vec{C}_{ijn} \in S^{l_n}$, $n = k,\dots,i+1$, such that

$$\vec{C}_{ij(i+1)} \cdot \vec{G}_{i+1} - x \cdot g_{ij} = \sum_{n=k}^{i} \vec{C}_{ijn} \cdot \vec{G}_n$$

(2) let $\vec{C}_i \in S^{l_n}$ such that $\vec{C}_i \cdot \vec{A}_i = 0$ for some $i,k < i \leq n$; then there exist vectors $\vec{C}_j \in S^{l_j}$, $j = k,\dots,i-1$ such that

$$\vec{C}_i \cdot \vec{G}_i = \sum_{j=k}^{i-1} \vec{C}_j \vec{G}_j \ .$$

The following generalization of a sufficient criterion for a basis to be a SZ-basis, the algorithms for the construction of a SZ-basis and the ample functions are due to Shtokhamer [7], where also the omitted proofs can be found.

5.9 Theorem: Let $M = \bigcup_{i=k}^{s} G_i \subset R$ with the properties of Lemma(5.8). Then M is an SZ-basis for U=(M) .

Remark : If S is a pr.i.d. then the vectors $\vec{C}_{ijn}$ of Lemma(5.8) (1) consist only of one element. For this case Szekeres gives a uniqueness condition for an SZ-basis in terms of conditions for the C_{ijn} and thereby gets an enumeration of all primitive ideals. His proof relies heavily on the fact that proposition (2) of Lemma(5.8) is trivial for pr.i.d.'s : $C_i \cdot A_i = 0$ implies $C_i = 0$ (S was assumed to be an integral domain). But this is not true for Noetherian domains where $\vec{C}_i$ and $\vec{A}_i$ may be vectors which consist of more than one element. Hence this part of Szekeres' construction cannot be generalized to Noetherian rings .

We assumed the existence of saf's for $(S,\sim,\leq,)$where $\sim$ is the congruence modulo an ideal and $\leq$ is a partial ordering. Let us first extend $\leq$ from S to R = S[x].

5.10 Definition : Let p , q ∈ R. Then

$$p < q \iff \begin{cases} \deg(p) < \deg(q) \\ \deg(p) = \deg(q) \ \& \ lc(p) < lc(q) \\ \deg(p) = \deg(q) \ \& \ lc(p) = lc(q) \\ \ \& \ red(p) < red(q) \end{cases}$$

and $p \leq q \iff p = q$ or $p < q$.

Let A be an ideal in S with basis $B = \{b_1,\dots,b_t\}$, f_A the corresponding saf. Then $f_A(s) - s \in A$ for all $s \in S$; hence there exist vectors $\vec{C} \in S^t$ such that $f_A(s) - s = \vec{C}\cdot\vec{B}$. Let g_A be a choice function for the set of all these vectors $\vec{C}$, i.e.

$g_A : S \to S^t$ and $f_A(s) - s = g_A(s)\cdot\vec{B}$, for all $s \in S$.
Now let $M = \bigcup_{i=k}^{m} G_i$ be an SZ-basis for the ideal U in R, let f_i be the saf for L_i, $g_i = g_{L_i}$. Let $G_i = \{g_{i1},\ldots,g_{il_i}\}$ and $B_i = \mathrm{red}(g_{i1}),\ldots,\mathrm{red}(g_{il_i})$.
Let $p = ax^N + q \in R, \deg(q) < N$. With these notations we define:

5.11 Definition :

$$(1)\quad f_j^*(p) = \begin{cases} p, & \text{if } N < j \\ f_j(a)\cdot x^N + f_j^*(q - g_j(a)\cdot B_j\cdot \\ \cdot x^{N-j}), & \text{if } N \geq j \end{cases}$$

for $j = k,\ldots,m$

$$(2)\quad F_U(p) = f_k^*\, f_{k+1}^* \ldots f_m^*(p).$$

f_j^* reduces p with respect to G_j by reducing successively its coefficients (beginning with the leading coefficient) with respect to L_j, but actually performing the the algebraic operations for the whole polynomials in G_j, not only for the leading coefficients. As in the previous section $F_U(p)$ may be considered as a generalized "remainder" of p after "division" by M.

5.12 Theorem : F_U is a saf for $(R,\sim,\leq)$, where $\sim$ is the congruence modulo U and $\leq$ is the ordering of definition(5.10).

If f_i has the property $f_i(0) = 0$ (which by condition (A2) for saf's implies that $f_i(s) = 0$ for all $s \in L_i$), then one can easily see $F_U(0) = 0$, too.

Let us now consider the computability aspect. If $M = \bigcup_{i=k}^{m} G_i$ is a given SZ-basis for U and if the f_i and g_i are computable then clearly F_U is computable.

So if the ideals A in S are detachable (which means that the predicate $s \in A$ is decidable) then all ideals U in S[x] are detachable, and with rather basic methods we have proven a result similar to one of Richman [14].
Since in our application we will have $S = S_1[x_1,\ldots,x_r]$, where S_1 is a pr.i.d., the saf's f_i will also require a special form of the bases for L_i (namely SZ-basis). We assume therefore that we have an algorithm A which constructs from a given basis of an ideal in S a suitable basis for the corresponding saf. If we have a finite set H of polynomials in S[x] we can apply algorithm A to their leading coefficients and perform the same algebraic operations for the corresponding polynomials. This algorithm is called ALGOR1. Further we assume the existence of an algorithm ALGOR2 which computes a finite basis for the S-module $V = \{\vec{v} \in S^n \mid \vec{v}\cdot\vec{B} = 0\}$ where $B = \{b_1,\ldots,b_n\}$ is a basis for an ideal in S. (Shtokhamer gives such an algorithm).

The following algorithm constructs from an arbitrary finite basis N of an ideal U a SZ-basis M for U.

Let $N = \bigcup_{i=k}^{m} H_i$, where the elements of H_i are of degree i; let $N_j = \bigcup_{i=k}^{j} H_i$, $j \geq k$.
For the output M we write similarly :

$$M = \bigcup_{i=1}^{m} G_i \quad , \quad M_j = \bigcup_{i=1}^{j} G_i .$$

We note that if $M = M_m$ is an SZ-basis, then so is M_j for all j with $1 \leq j \leq m$. According to definition (5.11) we denote by f_j^* the reduction function corresponding to G_j, and by F_j tha saf for M_j. We assume that $F_j(0) = 0$ and hence $F_j(p) = 0$ for all $p \in M_j$. If T is a set $\{p_1,\ldots,p_t\}$ of polynomials then

$$F(T) := \bigcup_{i=1}^{t} \{F(p_i)\} .$$

M ← SZBASE(N)

(1) if n = 0 then {M ← ALGOR1(N); return}.
(2) M_{n-1} ← SZBASE(N_{n-1}). [$M_{n-1} = \bigcup_{i=1}^{n-1} G_i$, a SZ-base]
(3) T ← $H_n \cup (x\cdot G_{n-1})$; G_n ← ALGOR1(T).
(4) D ← $F_{n-1}(f_n^*(T))$;
if D ≠ {0} then go to (7).
(5) A_n ← set of leading coefficients of elements of G_n;
V ← ALGOR2(A_n); T ← $\{\vec{v}\cdot\vec{G}_n \mid \vec{v} \in V\}$;
D ← $F_{n-1}(T)$.
(6) if D = {0} then {M ← $G_n \cup M_{n-1}$; return}.
(7) N_{n-1} ← $M_{n-1} \cup D$; H_n ← G_n; go to (2). ∎

We have
$F_{n-1}(T) = \{0\} \iff T \subset M_{n-1}$.
In step(4), $D = F_{n-1}(f_n^*(T)) =$
$= F_{n-1}(f_n^*(H_n)) \cup F_{n-1}(f_n^*(x\cdot G_{n-1}))$, hence $D = \{0\}$ implies $F_{n-1}(f_n^*(H_n)) = \{0\}$ and $F_{n-1}(f_n^*(x\cdot G_{n-1})) = \{0\}$. From $F_{n-1}(f_n^*(H_n)) = \{0\}$ follows that no polynomials are "lost" during the execution of the algorithm, i.e. that the output M is a basis for the same ideal U. From $F_{n-1}(f_n^*(x\cdot G_{n-1})) = \{0\}$ follows that condition (1) of Theorem(5.9) is satisfied.
If in step(6) D = {0} then condition(2) is satisfied and M is an SZ-base by Theorem(5.9). If $D \neq \{0\}$ then $D \not\subset M_{n-1}$, therefore the ideal generated by M_{n-1} is a proper subset of the ideal generated by $M_{n-1} \cup D$. Since R = S[x] is Noetherian there exists no infinite ascending chain of ideals and hence the algorithm terminates. We don't have a bound on the number of steps the algorithm SZBASE performs since there is no explicit bound on the length of an ascending chain of ideals in Noetherian rings. In [12] Seidenberg gives a bound, but it is a multirecursive function, and he furthermore works over a field (see also [13]); so his result doesn't solve the complexity problem for the above algorithm.

If $S = S_1[x_1,\ldots,x_{r-1}]$, where S_1 is a pr.i.d., the algorithm A mentioned above

will actually be a recursive call of SZBASE. As the recursion base we have $S = S_1$, and we must assume that we can compute saf's for S_1. If $S_1 = Z$, the suitable basis is the gcd of the elements of the given basis, and we may take f_1 or f_2 from section (3) as a saf for Z together with the corresponding ordering $\leq_1$ ($\leq_2$) .

6. Open questions

We have already mentioned that we don't know a general bound on the number of steps both algorithms perform. Further questions are :

a) how can Buchberger's algorithm be applied if the coefficient domain is a pr.i.d. instead of a field? The definition of an M-reduction has an obvious generalization, and also the algorithm terminates.

b) how does Shtokhamer's algorithm behave if all input polynomials are linear in total degree?

c) can one find conditions for an ideal to be contained in another ideal in terms of conditions for their bases (G-bases or SZ-bases) ?

References

[1] B.F. Caviness, On Canonical Forms and Simplification, J.ACM 17, April, 1970, (385-396)

[2] W.S. Brown, Rational Exponential Expressions and a Conjecture Concerning π and e, Amer. Math. Monthly 76, Jan., 1969, (28-34)

[3] J. Moses, Algebraic Simplification, a Guide for the Perplexed, Comm. ACM 14, Aug. 1971, (527-538)

[4] R. Loos, Toward a Formal Implementation of Computer Algebra, Proceedings of EUROSAM, Aug. 1-2, 1974, as SIGSAM Bulletin 8, Aug. 1974, (9-16)

[5] R.D. Jenks, The SCRATCHPAD Language, Proceedings of a SIGPLAN Symposium on Very High Level Languages, March 1974

[6] B. Buchberger, Ein Algorithmisches Kriterium für die Lösbarkeit eines Algebraischen Gleichungssystems, Aequationes mathematicae Vol. 4, 1970

[7] R. Shtokhamer, A Canonical Form of Polynomials in the Presence of Side Relations, Physics Department, Technion, Haifa, Israel, Technion-PH-76-25

[8] B.F. Caviness,P.L.Pollak, C.M. Rubald, An Existence Lemma For Canonical Forms in Symbolic Mathematics, Information Processing Letters (1971) 45-46

[9] A. Szekeres, A Canonical Basis for the Ideals of a Polynomial Domain, Amer. Math. Monthly 59, (1952) (379-386)

[10] G.E. Collins, Algebraic Algorithms, Prentice Hall, Chap. 2 (to appear)

[11] D.R. Musser, Algorithms for Polynomial Factorization, Comp. Science Dept. Technical Report No. 134, Sept. 1971, Univ. of Wisconsin

[12] A. Seidenberg, On the Length of a Hilbert Ascending Chain, Proc. of the Amer. Proc. Soc., Vol 29 (1971) (433-450)

[13] A. Seidenberg, Constructions in Algebra, Trans. of Amer. Math. Soc. , Vol. 197 (1974), 273-313

[14] F. Richman, Constructive Aspects of Noetherian Rings, Proc. of Amer. Math. Soc., Vol. 44 (1974), 436-441

[15] S.L. Kleiman, Computing with Rational Expressions in Several Algebraically Independent Variables, Bell Labs Tech. Report, Murray Hill, N.J.

[16] B. Buchberger, On Certain Bases of Polynomial Ideals, Universität Linz, Institut für Mathematik, Bericht Nr. 53, May 1976

Taking Advantage of Zero Entries in the Exact Inverse of Sparse Matrices

by
Paul S. Wang and Tadatoshi Minamikawa*

1. Introduction

An important use of a computer symbolic mathematical system lies in the area of matrix manipulation. This includes determinant [1,5] and linear equations [2,6,8]. Computing the inverse is one of the more complicated matrix operations. A principal difficulty in the exact inversion of a matrix is intermediate expression growth. This is particularly true if the given matrix contains a number of variables. The MACSYMA system [7] has been using an inversion scheme based on Gaussian elimination [6]. When the given matrix is sparse, it is desirable to have a way of taking advantage of the presence of the many zero entries. We used the technique of reduction to block triangular form [3]. The reduction is done by finding the strong components in a directed graph [4] associated with the given sparse matrix.

In section two we shall describe how this scheme is employed to compute the exact inverse. A proof will be given in section three to show that, as far as symbolic inversion is concerned, this algorithm gives all the "predictable" zero entries in the inverse. Section four contains timing data comparing this method with MACSYMA's Bareiss-type fraction free elimination scheme.

2. Taking Advantage of Sparsity

If the given matrix Q is sparse in the sense that it has many zero entries then its inverse may also have many zero entries. One obvious example of this situation is the triangular matrix with all zeros above or below the main diagonal. Substantial computation can be saved if the zero entries in the inverse are predicted so that they do not have to be computed. We describe in this section an algorithm based on this idea for inversion of sparse matrices. The algorithm hinges on the ability to efficiently find a way of permuting the rows and columns of Q so that it becomes upper block triangular.

Specifically if Q is an n x n sparse matrix, we try to reorder the rows and columns so that Q is transformed into $\tilde{Q}$ which has the following structure:

$$\tilde{Q} = \begin{pmatrix} \tilde{Q}_{11} & \tilde{Q}_{12} & \tilde{Q}_{13} & \cdots & \tilde{Q}_{1t} \\ 0 & \tilde{Q}_{22} & \tilde{Q}_{23} & \cdots & \tilde{Q}_{2t} \\ 0 & 0 & \tilde{Q}_{33} & \cdots & \tilde{Q}_{3t} \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & \tilde{Q}_{tt} \end{pmatrix}, \quad t \geq 1,$$

where $\tilde{Q}_{ij}$ is a matrix of dimension $n_i \times n_j$, $n_1 + n_2 + \ldots + n_t = n$. If Q is nonsingular then the matrices $\tilde{Q}_{ii}$ are nonsingular. The inverse $\tilde{Q}^{-1}$ can be computed rather easily from ${\tilde{Q}_{ii}}^{-1}$. The inverse $\tilde{R} = \tilde{Q}^{-1}$ has exactly the same block structure as $\tilde{Q}$. $\tilde{R}$ is computed block by block starting from the left-most column which is all zero except $\tilde{R}_{11} = {\tilde{Q}_{11}}^{-1}$. For the jth column of $\tilde{R}$ we have $\tilde{R}_{jj} = {\tilde{Q}^{-1}}_{jj}$

$$(1) \quad \tilde{R}_{ij} = -\tilde{R}_{ii} \sum_{k=i}^{j} \tilde{Q}_{ik}\tilde{R}_{kj}, \quad i < j$$

and, of course, $\tilde{R}_{ij} = 0$ for $i > j$. To obtain $\tilde{R}_{ii}$ one may use a Gaussian elimination scheme if $\tilde{Q}_{ii}$ is univariate. Otherwise computing cofactors seems promising. It is important to avoid GCD computations. Suppose $\tilde{Q}$ is rescaled to be fraction-free then care must be taken to avoid arithmetic with rational functions in computing $\tilde{R}_{ij}$. Let Δ_i be the determinant of $\tilde{Q}_{ii}$ and $R_{ii} = \Delta_i \tilde{R}_{ii}$. Then $\tilde{R}_{ii}$ can always be written as R_{ii}/Δ_{ii} with R_{ii} fraction-free. We see that $\tilde{R}_{ij}$ can be represented as

$$R_{ij}/(\prod_{k=i}^{j} \Delta_k), \quad i < j.$$

Therefore no GCD is needed in the summation given by (1).

To obtain Q^{-1} from $\tilde{Q}^{-1}$ is just a matter of undoing the row and column permutations that transformed Q to $\tilde{Q}$. The matrix Q is reducible if it can be rearranged in upper block triangular form. Otherwise, Q is irreducible. $\tilde{Q}$ is said to be in

*Paul Wang is with the Laboratory for Computer Science (formerly Project MAC) and Department of Mathematics, MIT, Cambridge MA 02139.

Tadatoshi Minamikawa is with Information Systems Laboratory, Toshiba Electric Co., Kawasaki 210, Japan.

Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation

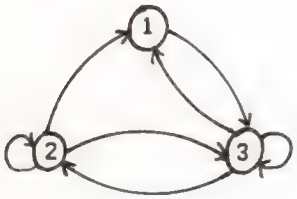
Otherwise, Q is irreducible. $\tilde{Q}$ is said to be in fully reduced form if its block structure can not be further refined.

Now let us consider the means of obtaining the desired block structure. A directed graph [4] g(Q) can be associated to the matrix Q. This graph has n nodes labeled 1 through n. The nodes are linked by directed edges representing nonzero entries of Q. An edge from node i to node j represents the nonzero entry q_{ij}. This edge is labeled q_{ij}. Only the nonzero entries of Q are represented in g(Q). A sequence of edges leading from node i to j is called a path from i to j. Two nodes i and j are said to be connected if there are two paths: one from i to j and the other from j to i. A subgraph is isolated if any pair of nodes in the subgraph are connected and no nodes outside the subgraph are connected to any inside, such isolated subgraphs are called strong components of g(Q). The strong components of g(Q) give rise to the block structure of Q. We denote by S_Q the number of strong components in g(Q).

The outcome of the above scheme is dependent on the given order of the rows and columns of Q. This means that a permutation of the rows and/or columns may result in an associated graph with more strong components and therefore lead to a refined block structure of Q. For example, if Q is given as

$$Q = \begin{pmatrix} 0 & 0 & a \\ b & c & d \\ e & f & g \end{pmatrix}$$

then g(Q) looks like



which has only one strong component. However, by interchanging the first and third rows of Q one would find t = 2. Indeed two is the maximum number of blocks Q has. As a matter of fact Q can always be fully reduced if nonzero elements are assigned on the main diagonal before constructing g(Q).

We define a matrix to be preassigned if it has no zero entries on the main diagonal. Any nonsingular matrix can be preassigned through row permutations [3].

Theorem 1. If the matrix Q is preassigned and R is any matrix obtained from Q by permuting rows and columns of Q then $S_Q \geq S_R$.

It follows immediately from theorem 1 that if Q is preassigned then $t = S_Q$ is maximum and $\tilde{Q}_{ii}$ are irreducible for i = 1, 2, ..., t. Thus, in our inversion algorithm, the given matrix is preassigned before its block structure is discovered using strong components.

The effect on g(Q) of interchanging row i and row j of Q is to replace all edges q_{ik} by q_{jk} and all q_{jk} by q_{ik}, or in other words, to move all edges originating from node i to node j and vice versa. Let $\pi = (i_1, i_2, \ldots, i_k)$ be a cyclic permutation and Q_π be a matrix obtained by applying π to the rows of Q. We can prove the following lemma.

Lemma. If Q is preassigned then $S_Q \geq S_{Q_\pi}$.

Proof. The lemma is true if we can show that connected nodes in g(Q) stay connected in $g(Q_\pi)$. The matrix Q being preassigned implies that the edges $q_{i_1 i_1}, q_{i_2 i_2}, \cdots, q_{i_k i_k}$ are all in g(Q). Therefore $q_{i_2 i_1}, q_{i_3 i_2}, \cdots, q_{i_k i_{k-1}}, q_{i_1 i_k}$ are in $g(Q_\pi)$. Because the permutation π only moves edges originating from these nodes among these nodes, it follows that connected nodes in g(Q) stay connected in $g(Q_\pi)$.

The above lemma holds if π is applied to the columns of Q instead of the rows of Q. The fact that any permutation can be decomposed into disjoint cyclic permutations proves theorem 1.

At this point let us list in summary form the steps for transforming Q into $\tilde{Q}$.

1) Preassign Q through row permutations. If this is not possible then Q is singular.

2) Construct g(Q) and identify the individual strong components by grouping together connected nodes in g(Q).

3) Arrange the strong components in such an order that there exists a path from the first component to the second, etc. Obtain an ordered list L of all nodes by putting the nodes in the first strong component first, etc. The ordering of nodes in the same component is immaterial. The list L provides a permutation of the list (1, 2, 3, ..., n) and gives both the row and the column ordering for obtaining $\tilde{Q}$ from Q.

3. Completeness of the Algorithm

Let us assume that we are interested in predicting the zero entries in the inverse that are due to the zero entries in the given matrix Q. This means that the prediction scheme has to disregard the specific values of the nonzero entries in Q other than noting that they are nonzero. We shall not consider here the introduction of additional zeros in Q by linear dependence checks on the rows and columns, etc. We define a zero in Q^{-1} to be unpredictable if it can arise only through cancellation.

Theorem 2. If Q is irreducible then there is no predictable zero entry in Q^{-1}.

Proof. The ij-th entry of Q^{-1} is $\alpha_{ij} = |Q_{ij}|/|Q|$ where Q_{ij} is obtained from Q^T by replacing the j-th

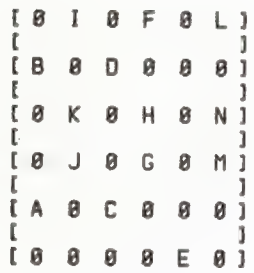
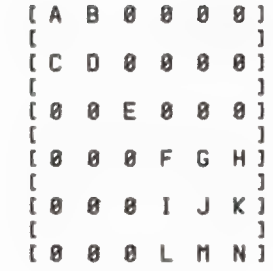
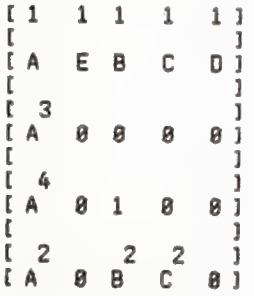
column by a column consisting of all zeros except a one in the ith row. Since Q is irreducible $g(Q^T)$ is strong. We shall show that there is at least one nonzero term in $|Q_{ij}|$ for any i, j. It then follows that no zero entry is predictable.

If $i = j$, entries on the main diagonal of Q_{ij} are all nonzero and we are done. For $i \neq j$ we consider the graph $g(Q_{ij})$ which is obtained from $g(Q^T)$ by removing all edges ending at node j and putting in an edge from node i to node j. Hence the nodes i and j are connected. There exists a path from i to j and back to i passing through a set of distinct nodes [i, j, k, l, ...]. Let π be the corresponding cyclic permutation (i, j, k, l, ...). Applying π to the rows of Q_{ij} produces a matrix with all nonzero entries on the main diagonal. It follows that $|Q_{ij}|$ has at least one nonzero term.

4. Timing Comparison

The inversions of eight sparse matrices are performed using two different methods available in MACSYMA: (a) a Bareiss-type fraction free elimination method and (b) the fraction free block triangular method. The timing is done on a DEC KL-10 computer which is estimated to be four and a half times faster than the KA-10 that MACSYMA used to be on. In some cases, the block triangular method is more than 10 times faster than method (a).

INPUT MATRIX	BLOCK TRIANGULAR FORM	INVERSION TIMES A/B (in milliseconds)	RATIO
$\begin{bmatrix} 0 & G & H & 0 \\ B & C & D & A \\ D & E & F & 0 \\ 0 & 0 & I & 0 \end{bmatrix}$	$\begin{bmatrix} A & B & C & D \\ 0 & D & E & F \\ 0 & 0 & G & H \\ 0 & 0 & 0 & I \end{bmatrix}$	153/94	1.62
$\begin{bmatrix} 2 & 2 & & 2 & & 2 \\ X & Y & 0 & U & 0 & Z \\ 4 & 4 & & & & \\ X & Y & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 5 & & & & & \\ X & 0 & 0 & 0 & 0 & 0 \\ 3 & 3 & & & & 3 \\ X & Y & 0 & 0 & 0 & Z \\ X & Y & 0 & U & V & Z \end{bmatrix}$	$\begin{bmatrix} 5 & 4 & 3 & 2 & & \\ X & X & X & X & X & 1 \\ & 4 & 3 & 2 & & \\ 0 & Y & Y & Y & Y & 1 \\ & & 3 & 2 & & \\ 0 & 0 & Z & Z & Z & 1 \\ & & & 2 & & \\ 0 & 0 & 0 & U & U & 1 \\ 0 & 0 & 0 & 0 & V & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$	520/264	1.97
$\begin{bmatrix} A & 0 & B & 0 \\ 0 & E & 0 & F \\ C & 0 & D & 0 \\ 0 & G & 0 & H \end{bmatrix}$	$\begin{bmatrix} A & B & 0 & 0 \\ C & D & 0 & 0 \\ 0 & 0 & E & F \\ 0 & 0 & G & H \end{bmatrix}$	316/95	3.32
$\begin{bmatrix} 0 & E & 0 & 0 & G & 0 \\ A & 0 & C & 0 & 0 & 0 \\ 0 & 0 & 0 & J & 0 & M \\ 0 & F & 0 & 0 & H & 0 \\ 0 & 0 & 0 & I & 0 & L \\ B & 0 & D & 0 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} A & B & 0 & 0 & 0 & 0 \\ C & D & 0 & 0 & 0 & 0 \\ 0 & 0 & E & F & 0 & 0 \\ 0 & 0 & G & H & 0 & 0 \\ 0 & 0 & 0 & 0 & I & J \\ 0 & 0 & 0 & 0 & L & M \end{bmatrix}$	2953/220	13.43

INPUT MATRIX	BLOCK TRIANGULAR FORM	INVERSION TIMES A/B (in milliseconds)	RATIO
		5356/342	15.66
		526/140	3.76
		438/223	1.96
		7993/708	11.29

Acknowledgement

We wish to express our thanks to Alex Doohovskoy and Joel Moses for many discussions and suggestions.

Work reported herein was supported by the Laboratory for Computer Science (formerly Project MAC), an MIT Interdepartmental Laboratory, sponsored by the United States Energy Research and Development Administration under contract E(11-1)-3070.

5. References

1. Gentleman, W. M., and Johnson, S. C., "Analysis of Algorithms, A Case Study: Determinants of Polynomials", Fifth Annual ACM Symposium on Theory of Computing, ACM, 1973, 135-141.

2. Griss, M. L., "The Algebraic Solution of Large Sparse Systems of Linear Equations via Minor Expansion," ACM Transactions on Mathematical Software, Vol. 2, No. 1, March 1976, pp. 31-49.

3. Gustavson, F., "Finding the Block Lower Triangular Form of a Sparse Matrix," RC5667, IBM Thomas J. Watson Research Center.

4. Haray, F., "Sparse Matrices and Graph Theory," Large Sparse Sets of Linear Equations, (Reid J., ed.) Academic Press, N.Y., pp. 139-150.

5. Horowitz, e., and Sahni, S., "On Computing the Exact Determinant of Matrices with Polynomial Entries", JACM, Vol. 22, No. 1, January 1975, pp. 38-50.

6. Lipson, J. D., "Symbolic Methods for the Computer Solution of Linear Equations with Applications to Flow-Graphs", Proceedings 1968 Summer Institute on Symbolic Mathematical Computation, IBM, 1969, 233-303.

7. Mathlab Group, MACSYMA Reference Manual, Laboratory for Computer Science (formerly Project MAC), MIT, Cambridge, Mass., November 1975.

8. McLellan, M. T., "The exact Solution of Systems of Linear Equations with Polynomial Coefficients", CACM, Vol. 20, 1973, pp. 563-588.

Completing nth Powers of Polynomials

by

Barry M. Trager
Department of EE/CS
Massachusetts Institute of Technology
Cambridge, Mass.

David Y.Y. Yun
Mathematical Sciences Department
IBM T. J. Watson Research Center
Yorktown Heights, New York

1. Introduction

A frequent exercise in high school algebra courses is completing the square of some given polynomial. The goal is to find terms involving only constants independent of the main variable, which when added to the given polynomial will result in a perfect square. As a typical example, $(x^2 + 4x + 3) + 1 = (x+2)^2$. The method for completing the square such as this one is often nothing more than applying the pattern matching abilities of students to the problem knowing the pattern $(x+y)^2 = x^2 + 2xy + y^2$. Here, we ask the question whether this problem can be generalized and whether there exists a constructive algorithm that replaces and extends the simple completion procedure of our high school days. The answer turns out to lie in the familiar process of computing polynomial remainder sequences (PRS) [Brown71].

2. Motivation and Derivation

Let $U(x)$ be a given polynomial in $I[x]$ with degree in x greater than zero where I is a unique factorization domain. A necessary condition for U to be a perfect nth power of some $V(x)$ in $I[x]$ (actually, for U to be divisible by any non-square-free polynomial) is that $\gcd(U, dU/dx)$ must not be 1. In fact, there must be a common divisor of U and dU/dx whose degree is at least as large as that of $V(x)$. Computing $\gcd(U, dU/dx)$ via the construction of polynomial remainder sequence $F_1 = U$, $F_2 = dU/dx$, $F_3,\ldots,F_k \neq 0$, we know that $\deg(F_k) \geq \deg(V) = \deg(U)/n$. Thus, as the PRS is being computed via division-with-remainder (or pseudo-division [Knuth69]) if any remainder (in our notation, it will be F_{k+1}) turns out to have degree less than $\deg(U)/n$ then that remainder must be identically zero.

Let c be an indeterminate representing a constant in I independent of x. Then the computation of PRS with $F_1 = U+c$ and $F_2 = d(U+c)/dx = dU/dx$ can be carried out as if c was a known quantity in I. When a remainder of degree less than $\deg(U)/n$ is encountered, it can simply be set equal to zero and solved for c. The constant so determined is added to U making

$$\gcd(U+c, d(U+c)/dx) \neq 1,$$

hence making it possible for $U+c$ to be a nth power.

Consider the previous simple example where $U(x) = x^2 + 4x + 3$. With $F_1(x) = U(x) + c$ and $F_2(x) = dU/dx = 2x + 4$, the first pseudo-divison step of the PRS process yields $4(U+c) = (dU/dx)Q + R$ where $Q = 2x + 4$ and $R = 4c - 4$. Since $\deg(R) = 0$ is already less than $\deg(U)/2 = 1$, we set R to zero and solve for c, getting $c = 1$. Indeed $U + 1$ is the perfect square that we are looking for.

A slightly more complex example will show the use of $\deg(U)/n$ as a terminating condition. It is not necessary to know n apriori, as we shall see later. However, prior knowledge of n does simplify the problem to an appreciable extent. Let $U = x^4 + 2x^3 - x^2 - 2x + 2$ and $n=2$, i.e., we want to determine whether U can be completed to a perfect square. With $F_1 = U + c$ and $F_2 = dU/dx = 4x^3 + 6x^2 - 2x - 2$, the first pseudo-division step of the GCD by PRS procedure gives $4^2 F_1 = F_2 Q_1 + R_1$ where $Q_1 = 4x + 2$ and $R_1 = -20x^2 - 20x + 36 + 16c$. We continue with the second division step, $(-20)^2 F_2 = F_3 Q_2 + R_2$ where $F_3 = R_1$, $Q_2 = -80x - 40$, and $R_2 = 320x + 320xc + 160 + 160c$.

Now $\deg(R_2) < \deg(U)/n$, so by setting R_2 to zero, we find $c=1$ and $\gcd(F_1, F_2)$ = primitive part of $R_1 = pp(R_1) = x^2 + x - 1$. In fact, $U + 1 = (x^2 + x - 1)^2$.

The next example shows that this construction solves an even more general problem. Consider the problem of completing $U = 2x^2 + 2x + 1$ to a perfect square. The first division step gives $16(U+c) = (dU/dx)Q + R$ with $dU/dx = 4x + 2$, $Q = 8x + 4$, and $R = 16c + 8$. Since $\deg(R) < \deg(U)/2$, we should set R to zero and solve for c (or $U+c$ could not be a square). Doing so we find that $c = -1/2$ (which is not in the same domain of consideration, I) and

$$U+c = 2x^2 + 2x + (1/2)$$

is not a square. However, letting $c = b/a$ so that each of a and b is in the correct domain, in I, we find

$$aU+b = (4x^2 + 4x + 2) + (-1) = 4x^2 4x + 1 = (2x + 1)^2$$

is a perfect square. Therefore, the problem of completing the nth power of $U(x)$ in $I[x]$ where I is only an integral domain and not a field can be generalized to that of finding a and b in I and $V(x)$ in $I[x]$ such that $aU(x) + b = V(x)^n$.

3. Algorithm

Input: a primitive polynomial $U(x)$ in $I[x]$ where I is a u.f.d., $\deg(U(x)) > 1$, and n an integer greater than 1.

Output: a, b in I and $V(x)$ in $I[x]$ such that $aU(x) + b = V(x)^n$ where $\deg(V(x)) \geq 1$.

(1) $F(x) \leftarrow U(x) + c$ where c is an indeterminant and $G(x) \leftarrow dU(x)/dx$.

(2) By pseudo-division, compute $Q(x)$ and $R(x)$ in $I[x]$ such that $mF(x) = G(x)Q(x) + R(x)$ where $m = lc(G(x))^{\deg(F) - \deg(G) + 1}$.

(3) If $\deg(R(x)) \geq \deg(U(x))/n$, then $F(x) \leftarrow G(x)$, $G(x) \leftarrow R(x)$, and go to (2).

(4) Set $R(x)$ to 0 and solve for c (= b/a) in the quotient ring of I (I/I). If either b or a (after cancelling common divisors) is not independent of x, then $U(x)$ is not completable to that nth power.

(5) $D(x) \leftarrow pp(G(x))$ (= $\gcd(U+c, dU/dx) = \gcd(aU + b, d(aU + b)/dx)$). With $aU(x) + b$ and $D(x)$, the process for finding the square-free decomposition of $aU(x) + b$ can be continued to get $U_j(x)$ in $I[x]$ and $n_j > 0$ such that $aU(x) + b = \prod_j U_j(x)^{n_j}$

(6) If n divides all n_j then $V(x) \leftarrow \prod_j U_j(x)^{(n_j/n)}$ otherwise $U(x)$ is not completable to a perfect nth power.

4. Existence and Uniqueness of Solution

If $U(x)$ in $I[x]$ is completable by constants a and b to powers of some $V(x)$ in $I[x]$, then the PRS starting from $U(x) + c$ and $dU(x)/dx$ must result in some non-trivial remainder or GCD. That is precisely the basis of the proposed algorithm for determining the existence of $V(x)$. To show the uniqueness of such solutions will require some additional machinery.

Theorem 1: If $U(x)$ is completable by constants to some non-trivial (>1) perfect power, then the constant is unique.

Proof: Let $U + c = V^n$ and $U + d = W^m$ where $c \neq d$ and V, W are non-trivial polynomials. Without loss of generality, can assume $n \geq m \geq 2$ then $\deg(V) \leq \deg(W)$. Subtracting the two relations, we get

$$V^n - W^m = c-d = e \neq 0 \quad \ldots (*)$$

We claim that V and W must be relatively prime. Assume otherwise, then there exists D in $I[x]$ such that D divides both V and W while $\deg(D) > 0$. But then D also divides $V^n - W^m = e \neq 0$ which is impossible. Now differentiate (*) and get

$$nV^{n-1} * (dV/dx) = mW^{m-1} * (dW/dx).$$

Since $\gcd(V,W) = 1$, we must have V divide dW/dx and W divide dV/dx. But $\deg(W) \geq \deg(V) > \deg(dV/dx)$, so that W can not divide dV/dx. Therefore, e must equal 0 or $c = d$. //

Corollary 1.1: If $V^n - W^n = c$, $n > 1$, then $c = 0$ and $V = W$ (up to units).

Proof: By Theorem 1, $c = 0$. Since $I[x]$ is a u.f.d., $V = W$. //

Corollary 1.2: If $V^n = W^m$ where $n > m$, then there exist S in $I[x]$ such that $V = S^{mk}$, $W = S^{nk}$ for some $k \geq 1$.

Proof: Without loss of generality, we may assume $\gcd(n,m) = 1$. Since otherwise, let $j = \gcd(n,m) > 1$ then $n = jn'$ and $m = jm'$ so $V^n = (V^{n'})^j = (W^{m'})^j = W^m$. By Corollary

1.1, $V^{n'} = W^{m'}$ and $\gcd(n',m') = 1$. Let $\deg(V) = d$, $\deg(W) = e$ then since $dn = em$ it must be that $d < e$. If $\gcd(V,W) = 1$, then as we have seen in proof of Theorem 1, W divides dV/dx, which is impossible. Thus let $D = \gcd(V,W)$, i.e., $V = DF$ and $W = DG$ where $\gcd(F,G) = 1$. But now $V^n = W^m$ implies $D^nF^n = D^mG^m$ or $D^{n-m}F^n = G^m$. So G^m divides D^{n-m} or $W^m = D^mG^m$ divides $D^mD^{n-m} = D^n$. Let $\deg(D) = f$, then since $dn = em \leq fn$ where $f \leq d$ it must be the case that $f = d$. That means $D = V$ so that V divides W or $W = VV'$ for some $V' \neq 1$ in $I[x]$ (if $V' = 1$, then $V = W$ contradicting $n > m$). Hence $W^m = V^mV'^m = V^n$ implies $V'^m = V^{n-m}$. Now the assumptions $\gcd(m, n-m) = 1$ and $\gcd(V', V) = 1$ can again be made, for otherwise some degree relation will be violated. We can now repeat the above argument on $V'^m = V^{n-m}$. If $m < n-m$ then we must have $V'^m = V^{n-2m}$, etc. until $n-qm < n$ and $V'^m = V^{n-qm}$ for some $q > 0$. Now $m > n-qm$ and the similar argument can be repeated with the corresponding replacements:

$$W \leftarrow V,\ m \leftarrow n-qm,\ V \leftarrow V',\ n \leftarrow m.$$

Note that this reduction process constitutes an integer remainder sequence process on the exponents n and m. But since $\gcd(n,m) = 1$, we eventually reach the situation $S^s = T^{pn+gm} = 1$ where T is a divisor of V, for some S in $I[x]$. Successively backtracking and we can deduce that S divides both V and W. Since $dn = em$ implies n divides e and m divides d, it must be the case that $V = S^{mk}$ and $W = S^{nk}$ for some $k \geq 1$. //

5. Termination Condition

The termination condition in Step (3) of Algorithm $\deg(R(x)) < \deg(U(x))/n$, came from the fact that if $U(x)$ is a perfect nth power of $V(x)$ then degree of the last non-trivial remainder of the PRS must be at least as large as $\deg(V(x))$. Thus no member of the PRS should have degree less than $\deg(V(x))$ and still be non-zero. But $\deg(V(x)) = \deg(U(x))/n$, hence the condition on the degree of $R(x)$ is terminating. The next step is to set $R(x)$ to zero and solve for the indeterminate c. Step (4) of the Algorithm does just that. However, we need a guarantee that the equation $R(x) = R(x,c) = 0$ can always be solved. Theorem 1 states that if such a solution c exists, it is unique. We might, then, expect that $R(x,c)$ is linear in c, so that c can be found uniquely. This was what we attempted to verify, but what we actually established was not only that c is linear in $R(x,c)$ but also a better terminating condition which requires no prior knowledge of the power n.

Consider the PRS $F_1 = U + c$, $F_2 = dU/dx$, $F_3,\ldots,F_k$, $F_{k+1} = R(x,c)$. Let $\deg(U) = d$ and define $\mathrm{loc}(F_i)$ to be the location of c in F_i or the highest power of x in F_i which has c in its coefficient. Then $\deg(F_2) = d-1$, $\mathrm{loc}(F_1) = 0$, $\mathrm{loc}(F_2) = 1$, and $\mathrm{loc}(F_3) = 0$. It is clearly true that $\mathrm{loc}(F_i) \leq \deg(F_i)$ for all $i=1,\ldots,k+1$. In general, the inequality would hold. So let us define

Termination Condition: $\mathrm{loc}(F_{k+1}) = \deg(F_{k+1})$.

We shall then determine the validity and utility of this condition.

Definition: A given polynomial $U(x)$ in $I[x]$ is said to be COMPLETABLE (by constants in I) if there exists a (unique) $c = b/a$ in I/I such that $aU + b$ is a perfect power of some $V(x)$ in $I[x]$.

Theorem 2: $\gcd(aU + b, d(aU + b)/dx) = \gcd(U + b/a, dU/dx)$.

Proof: Note that the first GCD is computed in $I[x]$ whereas the second GCD is computed in $(I/I)[x]$. But the proof follows from Gauss' Lemma and the fact that we can assume, without loss of generality, $\gcd(a,b) = 1$ so that $aU + b$ is primitive. //

This theorem allows us to compute GCD's in $I[x]$ and $(I/I)[x]$ interchangeably so long as the GCD computed is primitive and normalized [Brown71]. In fact, when we compute with $U + c$ for an indeterminate c, we are computing in $(I[c])[x]$ even though c may later be set to an element of I/I which will again put us back the domain of $(I/I)[x]$ and in turn to $I[x]$.

Returning to our PRS and the termination condition, it is known that $\deg(F_i) < \deg(F_{i-1})$ for $i = 3,\ldots,k+1$, i.e., the degrees of the PRS form a

monotonically decreasing sequence. A similar relation holds for locations of c in the PRS.

Lemma 3: $loc(F_i) > loc(F_{i-1})$ for all $i = 3,\ldots,k+1$.

Proof: It follows easily by considering polynomial division. //

This means that $loc(F)$ form a monotonically increasing sequence. Thus, as the PRS grows in length, it will eventually be the case that $loc(F_{k+1}) = deg(F_{k+1})$ for some $k > 2$. Indeed, we can establish:

Theorem 4: If $U(x)$ is completable then there is a $k \geq 2$ such that F_{k+1} satisfies the termination condition, $loc(F_{k+1}) = deg(F_{k+1})$.

Proof: Assume $loc(F_{k+1}) < deg(F_{k+1})$. Then, for the constant c that makes U completable to some power, $F_{k+1}(x,c)$ can not be identically zero. On the other hand, when the PRS reach the step $F_k = gcd(F_1,F_2)$, $F_{k+1}(x,c)$ must be identically zero. Therefore, F_{k+1} must satisfy the condition. //

What the theorem implies is that the PRS must terminate according to the condition and that the F_k when the condition is satisfied by F_{k+1}, is the GCD of F_1 and F_2 at the correct value of c.

At this point, we are ready to show that c appears linearly in all elements of the PRS when $U(x)$ is completable. We first need an additional lemma.

Lemma 5: For all $i = 3,\ldots,k$ (before the termination condition is satisfied by some F_{k+1}), $deg(F_1) \leq 2\, deg(F_i)$.

Proof: Since U is completable to some power < 1, the degree of $G = gcd(F_1,F_2)$ must not be less than that of F_1/G. The PRS terminates when $F_k = G$ and $deg(F_i) > deg(F_k)$ for all $i < k$, so $deg(F_i) \geq deg(F_1) - deg(F_k) \geq deg(F_1) - deg(F_i)$. //

Theorem 6: $loc(F_i) = loc(F_{i-1}) + deg(F_{i-2}) - deg(F_{i-1}) < deg(F_{i-1})$ for all $i=3,\ldots,k+1$.

Proof: By induction, for $i=3$, $loc(F_3) = 0 = loc(F_2) + d - (d-1) < deg(F_2)$. Assume the induction hypothesis holds for all $3 \leq i \leq j-1$. By Lemma 5, $deg(F_{j-1}) \geq deg(F_1) - deg(F_{j-1})$. Induction hypothesis implies that $deg(F_{j-2}) + loc(F_{j-1}) = deg(F_{j-3}) + loc(F_{j-2}) = \ldots = deg(F_3) + loc(F_2) < deg(F_3)$, we have $deg(F_{j-1}) > deg(F_{j-2}) + loc(F_{j-1}) - deg(F_{j-1})$ ...(*). Now consider the first step of dividing F_{j-2} by F_{j-1} via synthetic division, F_{j-1} is multiplied by x to the power $deg(F_{j-2}) - deg(F_{j-1})$. Thus the location of c in the first difference is exactly $loc(F_{j-1}) + deg(F_{j-2}) - deg(F_{j-1})$. This c can not be cancelled by any c from the later steps in the division, since the location of c later is smaller. By (*), the quotient polynomial is independent of c. Thus the remainder F_j will have c in that location, i.e.,

$$loc(F_j) = loc(F_{j-1}) + deg(F_{j-2}) - deg(F_{j-1}).$$

The inductive step is now complete, so is the proof. //

6. Generalizations

In view of the validity of the new termination condition, we can modify the algorithm to be more general as follows: (Note that n is now an output value rather than an input and only the steps with a "*" need to be modified.)

Input: a primitive polynomial $U(x)$ in $I[x]$ where I is a u.f.d., $deg(U(x)) > 1$.

Output: a, b in I, $V(x)$ in $I[x]$, and $n > 1$ such that $aU(x) + b = V(x)^n$ where $deg(V(x)) \geq 1$.

(1) $F(x) \leftarrow U(x) + c$ is an indeterminant and $G(x) \leftarrow dU(x)/dx$.

(2) By pseudo-division, compute $Q(x)$ and $R(x)$ in $I[x]$ such that $mF(x) = G(x)Q(x) + R(x)$ where $m = lc(G(x))^{deg(F) - deg(G) + 1}$.

(3*) If $deg(R(x)) > loc(R(x))$, then $F(x) \leftarrow G(x)$, $G(x) \leftarrow R(x)$, and go to (2).

(4) Set $R(x)$ to 0 and solve for $c(= b/a)$ in the quotient ring of I (I/I). If either b or a (after cancelling common divisors) is not independent of x, then $U(x)$ is not completable to any nth power.

(5) $D(x) \leftarrow pp(G(x))$ $(= gcd(U+c,dU/dx) = gcd(aU+b,d(aU+b)/dx))$.

With $aU(x) + b$ and $D(x)$, the process for finding the square-free decomposition of $aU(x) + b$ can be continued to get $U_j(x)$ in $I[x]$ and $n_j > 0$ such that

$$aU(x) + b = \prod_j U_j(x)^{n_j}.$$

(6*) Compute $n = gcd(n_j\text{'s})$. If $n = 1$ then U is not completable, otherwise then

$$V(x) \leftarrow \prod_j U_j(x)^{(n_j/n)}$$

Finally, we discuss several other generalizations. First, we point out that the GCD by PRS procedure (Steps (2) and (3)) can be modified to take advantage of the computationally more efficient methods, such as the Reduced or Subresultant PRS techniques [Collins67] [Brown 71]. The multivariate cases can also be handled by considering I to be a u.f.d. in all except the main variable x and appropriately extending the coefficient arithmetic operations. The last case worth mentioning is the situation where the input polynomial U(x) of I[x] is not necessarily primitive. We can certainly compute its content, C and primitive part, P(x), as a start. Then proceed to compute solutions (if they exist) a, b, n, and V(x), corresponding to P(x), and the solutions corresponding to U(x) are then simply $a \leftarrow ac^{n-1}$, $b \leftarrow bc^n$, n, $V \leftarrow CV$. Its justification is $aP + b = V^n \Rightarrow c^n aP + c^n b = c^n V^n \Rightarrow ac^{n-1} U + bc^n = (CV)^n$. This is an acceptable solution because we are completing U(x) to perfect powers with respect to the main variable x by finding quantities a and b that are independent of x.

7. Potentials

We have presented a general and constructive approach to the problem of completing arbitrary nth power of polynomials by constants. This proves the existence and uniqueness of solutions as well as the construction of them, gives a complete view to the problem and, we feel, puts the subject to rest.

Striving for such a complete solution to a problem was sufficient motivation for us to carry out the work. But we also believe that there are other interesting directions where our solution might find potential application or utilization.

As a general advantage of a polynomial completed to an nth power by a constant, it offers almost the similar advantage of simplicity in representation, (e.g., saving space for storage) as does some factored representations. Thus, it might increase intuitive understanding, facilitate easier qualitative analysis, and make possible faster, more efficient evaluation of the particular polynomial expression. All these can be accomplished at a relatively inexpensive cost of calculating polynomial remainder sequences (as opposed to factoring, for instance).

As a specific application of completing the square, we are reminded the reformulation of, say, a second order ordinary differential equation into an integral form susceptible to solution by optimization techniques. Such variational techniques for solving differential equations have found uses for completing higher "degree" "polynomials". The finite element method for approximating solutions to partial differential equations is, in its most fundamental form, nothing more than an application of the Ritz-Galerkin piece-wise approximation technique on the corresponding "square-completed" integral form.

Finally, even though we realized the fact after the completion of our work, we acknowledge a similar usage of the parametrized PRS discussed in [KNU69] (credited to Berlekamp) for speeding up Berlekamp's factorization algorithm. The goal is different from ours but the means appear to be quite similar. Unfortunately, Knuth offers no further insight to the method nor does he furnish a complete solution as we do.

References

[Brown71] "On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors", JACM, Vol. 18, No. 4, Oct. 1971, pp. 478-504.

[Collins67] "Subresultants and Reduced Polynomial Remainder Sequences", JACM, Vol. 14, No. 1, Jan. 1967, pp. 128-142.

[Knuth69] "THE ART OF COMPUTER PROGRAMMING, Vol. 2: SEMINUMERICAL ALGORITHMS, Addison-Wesley, Reading, Mass., 1969.

A Polynomial Decomposition Algorithm

by

David R. Barton and Richard E. Zippel

Laboratory for Computer Science, MIT
(formerly Project MAC)

Abstract

This paper presents an efficient, effective algorithm for decomposing a polynomial $f(x)$ into an irreducible representation of the form

$$f(x) = g_1(g_2(\ldots g_n(x) \ldots)).$$

This decomposition is used as an aid in solving high degree metacyclic equations in radicals and preconditioning polynomials for evaluation.

There are a number of algebraic manipulation systems which possess various routines for solving irreducible algebraic equations. For the most part, these systems are restricted to solving quadratic, cubic and quartic equations. One of the standard tricks they use when faced with an equation like:

$$x^{12} + 13\,x^8 + 17\,x^4 + 9 = 0$$

is to take the GCD (greatest common divisor) of the exponents and reduce the problem to:

$$y^3 + 13\,y^2 + 17\,y + 9 = 0 \qquad (1)$$

$$y = x^4.$$

This system of equations would then be solved by first computing the possible values for y in (1) and then calculating the corresponding values for x.

In this paper we generalize this technique by finding not only nested pure powers of x, but also nested polynomials in x. For example,

$$u(x) = x^6 + 6x^4 + x^3 + 9x^2 + 3x - 5$$

is irreducible over the rationals and is of degree greater than four, so it is not clear how to obtain its zeroes. Notice, however, that $u(x) = v(w(x)) = (v \circ w)(x)$ where

$$w(x) = x^3 + 3x$$
$$v(y) = y^2 + y - 5.$$

Since $w(x)$ and $v(y)$ are both of degree less than five, we can determine the zeroes of $u(x)$ by computing the zeroes of $v(y)$, α_1 and α_2, and then solving the equations $w(x) = \alpha_i$.

First we need some definitions. A *decomposition* of a polynomial $f(x)$ over a field $\mathfrak{F}$ into polynomial *components* $g_i(x)$ is an identity of the form:

$$f(x) = (g_1 \circ g_2 \circ \ldots \circ g_n)(x).$$

Notice that since any linear polynomial has an inverse which is also a polynomial (the inverse of $ax+b$ being $x/a-b/a$), there are an infinite number of *trivial decompositions* of any polynomial $f(x)$: i.e. ones where all but one of the $g_i(x)$ are linear. A component is *indecomposable* if it is of degree greater than one and has no non-trivial decomposition. A *complete decomposition* is one where all the $g_i(x)$ are indecomposable.

Complete decompositions are not unique. Consider the following examples:

$$u(x) = x^6 + 6x^4 + x^3 + 9x^2 + 3x - 5$$
$$= (x^2 + x - 5) \circ (x^3 + 3x) \qquad (2)$$
$$= (x^2 + x - 5) \circ (x + 2) \circ (x - 2) \circ (x^3 + 3x)$$
$$= (x^2 + 5x + 1) \circ (x^3 + 3x - 2) \qquad (3)$$

$$x^{12} = x^2 \circ x^2 \circ x^3$$
$$= x^2 \circ x^3 \circ x^2$$
$$= x^3 \circ x^2 \circ x^2$$

We define the Chebyshev polynomials $t_n(x)$ by:

$$t_n(\cos u) = \cos nu.$$

Then

$$t_{mn}(x) = (t_m \circ t_n)(x) = (t_n \circ t_m)(x).$$

Clearly the complete decompositions of each polynomial, though not identical, are similar. We consider the two decompositions

$$f(x) = (g_1 \circ g_2 \circ \ldots \circ g_n)(x)$$
$$= (h_1 \circ h_2 \circ \ldots \circ h_n)(x)$$

equivalent if there exist linear $\lambda_1(x)$, $\lambda_2(x)$, . . . , $\lambda_{n-1}(x)$ such that

$$g_i(x) = (\lambda_{i-1} \circ h_i \circ \lambda_i^{-1})(x).$$

Thus the two complete decompositions of $u(x)$, (2) and (3), are equivalent. Also, if two decompositions are equivalent to ones that differ only in the order of cyclic (x^n) and Chebychev polynomials, we call them equivalent too. Thus all the examples so far illustrate distinct but equivalent complete decompositions.

Ritt [3] showed that all complete decompositions of any given polynomial are equivalent. Thus, if the main application of this procedure is the solution of univariate polynomial equations, then the multiplicity of equivalent decompositions is not important. Each decomposition describes a tower of algebraic extensions. If the entire tower is solvable any subtower must also be solvable. Consequently if there exists a solvable decomposition, all decompositions are solvable. Therefore, we will be content to find any particular complete decomposition of a given polynomial. Notice that by introducing the proper linear polynomials $\lambda_i(x)$, we can specify the leading coefficient and constant term of each component other than the first.

The algorithm we present accepts a polynomial $f(x)$ and returns a complete decomposition in the form of a list of polynomials $g_i(x)$. The fundamental result upon which our algorithm rests is a special case of results due to Fried and MacRae [2] (see also [3] [1]). Theorem 2.3 of [2] states: "Let k be an arbitrary field and let $f(t)$, $g(t)$, $f_1(t)$, $g_1(t)$ be polynomials in $k[t]$. Then $f_1(x) - g_1(z)$ divides $f(x) - g(z)$ in $k[x,z]$ if and only if there exists a polynomial $F(t)$ in $k[t]$ such that $f(t) = F(f_1(t))$ and $g(t) = F(g_1(t))$." In particular, if $f(x) = g(x)$ then $f_1(x) = g_1(x)$. Thus, if $f(x)$ possesses a decomposition, $f(x) = (g \circ h)(x)$, then $h(x) - h(y)$ divides $f(x) - f(y)$. Conversely, if $f(x) - f(y)$ has a factor of the form $h_1(x) - h_2(y)$, then $h_1(x) = h_2(x) = h(x)$ and there exists a $g(x)$ such that $f(x) = (g \circ h)(x)$.

From this theorem it is clear how to calculate an $h(x)$ (where $f(x) = g(h(x))$). We factor $f(x) - f(y)$:

$$f(x) - f(y) = a_1(x,y)\, a_2(x,y) \ldots a_r(x,y)$$

where the a_i are irreducible polynomials in $\mathfrak{F}[x,y]$. We look for some product of the $a_i(x,y)$ which is of the form $h(x) - h(y)$ (a *separated polynomial*). By the theorem, for each such $h(x)$ there is a corresponding $g(x)$, such that $f(x) = (g \circ h)(x)$.

There are several points worth mentioning here. First of all, $x-y$ divides $p(x)-p(y)$ for any polynomial $p(x)$ (since $x-y$ divides x^n-y^n for all positive n), so in particular $x-y$ divides $f(x)-f(y)$. Therefore, instead of factoring $f(x)-f(y)$ we need only factor $(f(x)-f(y))/(x-y)$, and add $x-y$ to the list of $a_i(x,y)$. Also we know $x-y$ divides $h(x)-h(y)$, so we will only consider products of the $a_i(x,y)$ which include $x-y$ as a factor when searching for a separated polynomial $h(x)-h(y)$.

Secondly, we must ignore trivial decompositions, i.e. ones where $g(x)$ or $h(x)$ is linear. Thirdly, since we may specify the leading coefficient and constant term of $h(x)$, we agree to make $h(x)$ satisfy:

(i) $h(0) = 0$ (i.e. $h(x)$ has no constant term),
(ii) the leading coefficient of $h(x)$ is positive, and
(iii) $h(x)$ is primitive.

(Condition (iii) is relevant only if we wish to work over a ring. Over a field we can require $h(x)$ to be monic.)

While assumptions (i) and (ii) are merely for convenience, (iii) is very important. If $f(x)$ is a polynomial over a *ring*, (iii) implies, by a slight modification of the proof of our fundamental result (that $h(x)-h(y)$ divides $f(x)-f(y)$ if and only if $f(x) = (g \circ h)(x)$ for some $g(x)$) given in [1], that the coefficients of $g(x)$ and $h(x)$ are also in that ring. Thus we need not use rational number or rational function arithmetic when working over the integers or polynomial rings.

Finally, there may be more than one non-trivial factor of $f(x)-f(y)$ of the form $h(x)-h(y)$. Since we are only interested in obtaining one complete decomposition of $f(x)$, we could pick any one of these $h(x)$, find the corresponding $g(x)$ (as explained below), and recursively decompose $g(x)$ and $h(x)$. However, this will lead to additional multivariate factorizations, which are relatively costly.

Instead, we can work with the set of possible $h_i(x)$ that we now have. If we pick $h(x)$ to be a *minimal* $h_i(x)$, i.e. one such that there is no other $h_j(x)$ with $h_j(x)-h_j(y)$ dividing $h(x)-h(y)$ non-trivially, then $h(x)$ must be indecomposable, by our fundamental theorem. We then look at the $h_i(x)$ for which $h(x)-h(y)$ divides $h_i(x)-h_i(y)$ non-trivially, and we pick a minimal $h'(x)$ from this set. It will completely decompose as $(g \circ h)(x)$, after which we find a minimal $h''(x)$ with $h''(x) = (g' \circ h')(x)$. Continuing in this manner we obtain a complete decomposition without any more factorizations:

$$f(x) = (g^{(n)} \circ \ldots \circ g' \circ g \circ h)(x).$$

Now suppose we have $f(x)$ and $h(x)$ and we want a $g(x)$ such that $f(x) = (g \circ h)(x)$. In the general case, we would do a series of divisions with remainder, essentially calculating a p-adic expansion of $f(x)$ at $h(x)$:

$$\begin{aligned} f(x) &= q_1(x)\, h(x) + r_0(x) \\ q_1(x) &= q_2(x)\, h(x) + r_1(x) \\ &\ldots \\ q_s(x) &= 0\, h(x) + r_s(x) \end{aligned} \qquad (4)$$

with the degree of $r_i(x)$ less than the degree of $h(x)$ for all i. Then

$$f(x) = r_0(x) + r_1(x)\, h(x) + \ldots + r_s(x)\, h(x)^s$$

and

$$g(t) = r_0(x) + r_1(x)\ t + \ldots + r_s(x)\ t^s.$$

But since $g(t)$ must be a polynomial in t the $r_i(x)$ are constants ($r_i(x) = r_i$).

Since in our case $h(0) = 0$, we may compute the r_i using only exact divisions, as follows. Substituting $x=0$ in (4) yields

$$f(0) = q_1(0)\,h(0) + r_0(0) = r_0$$

$$q_1(0) = q_2(0)\ h(0) + r_1(0) = r_1$$

$$\ldots$$

$$q_s(0) = 0\ h(0) + r_s(0) = r_s.$$

Therefore

$$q_1(x) = [f(x) - f(0)]/h(x)$$

$$q_2(x) = [q_1(x) - q_1(0)]/h(x)$$

$$\ldots$$

$$q_s(x) = [q_{s-1}(x) - q_{s-1}(0)]/h(x)$$

This completes the discription of the decomposition algorithm.

As an example, let us again consider

$$u(x) = x^6 + 6x^4 + x^3 + 9x^2 + 3x - 5.$$

We proceed as follows:

$$\begin{aligned} u(x) - u(y) &= (x^6 + 6x^4 + x^3 + 9x^2 + 3x - 5) \\ &\quad - (y^6 + 6y^4 + y^3 + 9y^2 + 3y - 5) \\ &= (x - y)\ (x^2 + xy + y^2 + 3) \\ &\quad (x^3 + 3x + y^3 + 3y + 1) \\ &= (x^3 + 3x - y^3 - 3y) \\ &\quad (x^3 + 3x + y^3 + 3y + 1) \\ &= (w(x) - w(y)) \\ &\quad (x^3 + 3x + y^3 + 3y + 1), \end{aligned}$$

where $w(x) = x^3 + 3x$. Now

$$\begin{aligned} q_1(x) &= [u(x) - u(0)]/w(x) \\ &= [x^6 + 6x^4 + x^3 + 9x^2 + 3x]/[x^3 + 3x] \\ &= x^3 + 3x + 1 \\ q_2(x) &= [q_1(x) - q_1(0)]/w(x) \\ &= [x^3 + 3x]/[x^3 + 3x] \\ &= 1 \end{aligned}$$

so $v(x) = x^2 + x - 5$, as we claimed previously.

The question of how often such a decomposition is possible naturally arises. Notice that the product of the degrees of the components is simply the degree of the original polynomial. In particular, if the original polynomial is of prime degree, there is no non-trivial decomposition. Let us consider the simplest non-trivial case, i.e. degree four. Let

$$\begin{aligned} f(x) &= x^4 + a\ x^3 + b\ x^2 + c\ x + d \\ &= (x^2 + r\ x + s) \circ (x^2 + t\ x) \\ &= x^4 + 2\ t\ x^3 + (t^2 + r)\ x^2 + r\ t\ x + s, \end{aligned}$$

or

$$a = 2\ t$$

$$b = t^2 + r$$

$$c = r\ t$$

$$d = s,$$

which has a solution if and only if $a^3 - 4ab + 8c = 0$.

This approach generalizes, though for degrees with more factors, there will be more possible decompositions. Assume we want to decompose a monic polynomial of degree n into two components of degrees p and q, $n = p\ q$. There are $d(n)$ ways to do this, where $d(n)$ is the number of divisors of n. Each leads to $p\ q$ equations in $p + q - 1$ unknowns, defining a variety of codimension $p\ q - p - q + 1 = (p - 1)\ (q - 1)$.

A perfect power of a linear polynomial is irreducible by our definition, but in solving equations it is useful to note such a situation. It is easy to recognize this, however. Assume $f(x) = (ax+b)^n+c$. Let $g(x)$ be the primitive part of the derivative of $f(x)$ with respect to x. The remainder of $f(x)$ when divided by $g(x)$ must be c. The greatest common divisor of $f(x)-c$ and $g(x)$ is $(ax+b)^{n-1}$, assuming $gcd(a,b)=1$.

Finally we note some potential applications of our algorithm. In addition to solving polynomial equations it may be used for pre-conditioning polynomials which are to be repeatedly evaluated, i.e. in coding theory and numerical analysis. Also, the decomposition of a polynomial gives the complete lattice structure of the subfields of the genus 0 curve $y=f(x)$.

We wish to thank Barry Trager for his help and comments. This work was supported in part by United States Energy Research and Development Administration contract #E(11-1)-3070.

References

[1] Evyatar, A. and Scott, D.B., "On Polynomials in a Polynomial," Bull. London Math. Soc. 4, 176-178 (1972).

[2] Fried, M.D. and MacRae, R.E., "On Curves with Separated Variables," Math. Ann. 180, 220-226 (1969).

[3] Ritt, J.F., "Prime and Composite Polynomials," Trans. AMS 23, 51-66 (1926).

ASHMEDAI and a Large Algebraic Problem

M. J. Levine*
Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213
Department of Physics

and

R. Roskies†
University of Pittsburgh, Pittsburgh, Pennsylvania 15260
Department of Physics and Astronomy

This paper is dedicated to the memory of R. C. Perisho, colleague and collaborator.

Abstract

We give an overview of ASHMEDAI, an algebraic manipulation language developed by M. J. Levine. Many of its features are determined by the large physics problems we want to solve. These demand both rapid execution times and efficient storage management. We achieve rapid execution by not investing symbols with their full algebraic significance. For example, this eliminates the need for built-in rational function capabilities or for a very general integration routine, but does require more user intervention. We overcome the storage problem by using internal data structures and a processing scheme which allow intermediate expressions to be virtual and which allow the use of auxiliary storage devices without sacrificing execution efficiency. We point out how parallel processing might speed up execution even more and suggest other ways for improving the system. We also emphasize the importance of verification procedures, and detail some of ours.

INTRODUCTION

In this paper, we wish to discuss the algebraic system ASHMEDAI,[1] which has been used to great advantage in large scale analytic calculations in theoretical physics. To put the discussion in perspective, we begin with a very brief historical account.

ASHMEDAI is written by one of us (MJL). It arose initially as a set of limited purpose algorithms to do many of the tedious straightforward operations arising in Weak Interaction and Quantum Electrodynamic (QED) calculations. Levine then realized that these algorithms could naturally be extended to become a more general algebraic language. Through the course of several implementations, the package was a fairly austere machine language, Fortran-callable polynomial manipulation scheme together with special features to handle the specialized structures (Dirac matrices and Lorentz 4-vectors) arising in QED. Shortly before the collaboration with Roskies on computing the magnetic dipole moment of the electron, Levine instituted many changes in the program. Fortran was adopted as the source language to facilitate writing, debugging, modifying and disseminating the code. A very simple but very much improved user interface was developed to facilitate usage of the system. Numerous capabilities of the machine language versions were discarded because they did not seem necessary. Similarly, some algorithms were degraded to facilitate coding even at the expense of execution time efficiency if that seemed likely to speed up the work as a whole. However, as a result of difficulties encountered in the magnetic moment problem, many changes have been made which greatly increased the efficiency of the program and expanded its capabilities. Sometimes this involved reinstating older, more efficient procedures, and sometimes entirely new procedures were developed.

We feel very strongly that being 'led by the problem' is one important technique to be used in the development of algebraic systems. This

presentation is made not so much to explain the system as to point out to you who spend more of your time on such things those types of capabilities which we have found to be vital for some serious calculations.

Since so many of the program capabilities were inspired by the difficulties of the magnetic moment calculation, we begin by briefly discussing this problem, showing what kinds of execution features it requires. Then we discuss the program itself, how it looks to the user, its basic structure, its operating philosophy, some aspects of its storage management, some of its special features, and ways in which it can be improved. We then devote a section to checking techniques. This aspect of algebraic programming is rarely discussed, but is very important and involves a great deal of effort. We end with conclusions.

THE PHYSICS PROBLEM

Physically, the problem is to predict the dipole magnetic moment of the electron. This is the strength of the interaction of a stationary electron with a constant magnetic field. The experimental accuracy of this measurement is about 1 part in 10^9. The calculation required to match this precision is of a 6th order term in the perturbation theory expansion of Quantum Electrodynamics.[2]

The spin and relativistic behavior of the electron require the use of some special algebraic structures like Dirac matrices, Lorentz 4-vectors and the metric tensor. All programs used in this area can perform the algebraic manipulations these structures require. This aspect will not be discussed any further.

In addition to these special operations we use a wide variety of general symbolic substitutions. These effect the replacement of some algebraic expressions by others under certain circumstances. The simplest sort is a change of variables: whenever a certain quantity appears it is replaced by some expression in other quantities (indeterminates). The calculation also requires that we perform various simplifications, use the properties of common functions and simple rational expressions, do integrals, evaluate limits and perform differentiation.

Rather than build programs to perform these operations algorithmically on very general data structures we have used an augmented substitution capability operating upon polynomials in many indeterminates. Complicated or compound factors including inverses of other indeterminates or of simple polynomials, and simple functions of indeterminates are treated as separate indeterminates. The basic operations used in simplification which might otherwise be derived by rational function algebra or as properties of common functions are then implemented by performing specific substitutions. It must be emphasized that these indeterminates are simply names. They have no properties within the system other than those given in explicit substitution statements. Further, these statements are never global in nature. They are used only when and where specifically requested by the user. (Whatever we may be thinking, when performing algebra by hand, the external manifestation is the replacement of some pencil marks by others.)

For example, the partial fraction simplification

$$1/(x(1+x)) = 1/x - 1/(1+x)$$

corresponds to the substitution

$$IX \cdot IXP = IX - IXP$$

where $IX \sim 1/x$ and $IXP \sim 1/(1+x)$. This approach avoids the large execution times and coding problems associated with rational function algorithms, but it does require the user to have a detailed understanding of the structure of possible denominators. In a similar vein, integrals are performed by replacing integrand by integral. We take limits by replacing quantities by their limiting values. Differentiation is done by replacing the product of the 'derivative operator' and a given function by the derivative function (all represented by simple indeterminates). Simple extensions mimic the chain rule for differentiation and the rule for integration by parts.

We believe that this approach is efficacious on many counts. Less code need be written. By using the same code for many operations we save core to be used for very large data structures. Our more constrained approach is readily extended in new circumstances (e.g., differentiation of a new function by addition of a single substitution). Finally, it executes considerably more quickly than code which must be continually on the alert for a wide variety of possible alternatives.

That questions of efficiency in execution time and storage space are relevant can be seen by reviewing the history and complexity of this calculation. Lower order terms in the perturbation theory were evaluated using hand techniques. They required several man-years of work by many groups to be properly completed. Using comparable techniques, the current calculation is at least 1000 fold more difficult. Most previous work in the order we have tackled has been done numerically. That approach was sufficient, temporarily. It seems unable to provide the requisite accuracy in reasonable times. As an example of the magnitude of the 6th order work, consider the part of the calculation (a single Feynman graph, 1 of 40) we most recently completed.[3] The times spent in Dirac matrix manipulation, partial fraction simplification and integration were, 3 minutes, 10 hours and 20 minutes, respectively. The input expression required a few lines of type. The basic tables of integrals and the basic identities numbered a few hundred items. The combined sets of input data, commands and comments are contained in a few thousand cards, which as usual are largely blank. Individual intermediate expressions ranged up to several tens of thousands of terms. Altogether some 10^7 different terms were processed. Yet the final answer was a simple 8 term polynomial with rational coefficients in 4 basic transcendental numbers.

THE COMPUTING SYSTEM

All of these manipulations were performed using ASHMEDAI, a simple general purpose algebraic manipulation program. ASHMEDAI is written entirely in Fortran V. As we run it on a Univac 1108 it is a single overlay of about 24K words of program and uses another 28K words for general storage of

data. This storage is allocated dynamically as the calculation develops. Although our machine affords us no permanent on-line storage, we do use about half a million words of drum storage at execution time for sorting and saving very large data structures.

Some program features are simply due to the environment in which we use the system. We run in a single user, batch mode without data files. Thus the program response to detected errors is minimal. Without file storage and swapping facilities program size must be kept down and overlaying frees core only at the expense of drum space.

ASHMEDAI also runs on many PDP-10 and IBM 360-370 machines. There it can be used interactively. Conversion is no worse than for any large Fortran program such as those used in high energy physics data processing. It is being used for a variety of field theory and perturbation theory applications. Most of these applications are in high energy physics.

ASHMEDAI implements interpretively a modest algebraic processing language. From the user's viewpoint, ASHMEDAI differs from most programming languages. The program is comfortable and easy to use for problems which flow in one direction -- problems which start with some input expression, require substitutions and collection of terms, more substitutions, a new collection of terms and so on. To the uninitiate it is remarkable how many problems can be handled this way. Conditional branching, looping structures, and the presence of several expressions simultaneously are all possible, but require more user intervention than in some other programming languages. The user also has more decisions to make, such as where to invoke certain substitutions and where to collect terms. The advantage is greater efficiency, and small storage requirements. With the drum utilization technique to be described later, we are able to run a powerful program interactively on a PDP-10 in a 24K partition.

The input character stream contains commands which cause certain operations to be performed or which modify the way in which they are performed. That character stream also contains input data structures and comments. The basic input data structure is the 'term' or product of constants and indeterminates. These terms are combined to form 'expressions' and 'identities' according, largely, to the usual rules for Fortran 'arithmetic expressions'. The entire input character stream is scanned from left to right, one character at a time. This stream is derived principally from data cards. Commands can cause characters to be taken from other devices or from data files. Elaborate 'textual functions' which are similar to the macros used in assembly languages, and conditional commands can be used to augment the input character stream.

Output from the program contains a variety of information much of which can be suppressed by appropriate commands. There is a running commentary containing images of cards read, characters derived from textual functions, names of commands recognized, names of symbols created, the elapsed time, the amount of core available and counts of the number of terms which have been processed. Intermediate results and temporary data structures can be printed, directed to card, tape or data files for reuse by other ASHMEDAI runs. Results can be printed for reading or punched for later use in Fortran programs.

Explicit commands are used to cause operations to be performed; operations are usually not triggered simply by the appearance of their related data structures. For example, the appearance of a new substitution need not cause that substitution to be implemented. Expressions and identities can be input and left in core, inert and unused. A few commands cause operations to be performed on an expression in its entirety; most commands refer to a 'one term at a time' processing mode which can be tailored to make the best use of storage at little expense in time.

Those commands in ASHMEDAI which operate on an expression in its entirety are used to change the structure of the expression. Such changes are intended to facilitate either Fortran numerical evaluation of the expression or subsequent symbolic work. For example a polynomial in 2 variables (x,y) each occurring with up to the 99th power might contain 10^4 terms. Complicated operations involving only one of the variables(e.g. x) need only be performed 100 different ways. Operating on the whole expression would require 100 times the work. By restructuring the expressions into a polynomial in x with coefficients which are polynomials in y and operating only upon the polynomial in x, such extravagant wastes of time can be avoided.

Because of its utility in large calculations we will now describe the 'one term at a time' processing scheme and the 'control stack' of commands and pushdown list of intermediate, partial results used in its implementation. After that, we will describe a use of auxiliary storage which allows collecting terms and then reprocessing them in this 'one at a time' fashion even though the expressions will not fit in core.

Operations involving only single terms of larger polynomials arise naturally in these problems. Replacements wherein the sought expression is not a single term require very time-consuming multiple scans of expressions. We do without these. Differentiation is also performed on single terms. Integration can often be performed term by term. An exception occurs when recognition of a polynomial 'integrating factor' can greatly facilitate an integration. Aside from the simple distributive rule, all commonly used Dirac matrix and Lorentz 4-vector operations are also performed only on products of simple factors.

The utility of the 'one term at a time' approach rests both upon a recurring feature of these calculations and upon a disparity between the numbers of terms which time allows us to process and which space allows us to store. (In less than 1 minute we can process more terms than will fit in memory.) Consider an initial polynomial P_1, the operation $\emptyset_{12}$ which takes it into polynomial P_2 and the operation $\emptyset_{23}$ which takes that into polynomial P_3. Often, P_1 readily fits in the machine, P_2 exceeds its capacity yet P_3 once again fits. Especially if P_2 is subject to little simplification by collecting like terms, we profit by never having all of P_2 exist at any one time. Situations of this kind occur over and over again in these calculations.

This storage problem for P_2 may be completely avoided. We simply take one term of P_1, perform

ϕ_{12} upon it to get $\tilde{P}_2$, a part of P_2, and immediately perform ϕ_{23} on that to produce part of P_3. Before taking the next term from P_1, we destroy the part of P_2 which we have created but no longer need. In this way P_2 becomes virtual: it never exists in its entirety at one time. We carry this scheme to its extreme by making even $\tilde{P}_2$ virtual if it contains more than a single term. Thus at a given instant, the immediate input to a routine is kept to a single term. Its output is kept to a single term and whatever information it needs to resume work at a later time. The list of operations to be performed is the 'control stack'. There is also the list containing the current input terms and the 'impure data' needed for resuming processing. These two structures are cross linked and the latter changes, dynamically, as the processing continues.

The operations in a control stack are not executed as they are extracted from the input character stream. Recognition of one of the commands which can initiate a control stack causes ASHMEDAI to start queuing commands for this stack. Such a command must produce one term at a time from some data structure. Subsequent requests for substitutions or other 'one term at a time' operations are included in the growing control stack. Recognition of a command to collect like terms or somehow dispose of terms causes ASHMEDAI to terminate the control stack. Then the stack is 'executed'.

Terms start flowing from the initial to the final real expressions through the worker routines in 'one at a time' fashion. This continues until the final real expression is complete. The temporary storage possibly including the input expression, some identities and the control stack is relinquished and commands are again taken from the input character stream. Most calculations require many different control stacks for their completion.

Although this mode of operation places a load upon the user, we know of no reasonable alternative. Some intermediate expressions are too large for storage and must remain virtual. Some collapse very little when like terms are collected. For these, even though they may fit in storage, the time spent collecting like terms would be wasted. They, too, should remain virtual. However, when an intermediate expression is not prohibitively large and when it will collapse appreciably, one should collect like terms and create a real expression. Unfortunately, it is usually not clear ahead of time when to use each approach. As a result ASHMEDAI leaves it up to the user. He must then decide by understanding the structure of the problem or by trial and error.

Since we often find it necessary to collect terms on expressions which would vastly exceed our core memory, ASHMEDAI does this operation using the combination of a very efficient comparing algorithm and about half a million words of drum storage. The job of the comparing algorithm is to take each term as it is produced in the 'one at a time' fashion and to compare it with the terms it has already collected in core. If the new term is like one already in the collection, their coefficients are added. If the new term has no counterpart in the collection, it is appended to the collection. If there is no room in memory for this new term, the collection is declared closed and the new term is written out on drum. Once the collection in core is declared closed, no new terms are appended to it even though more core may later become available. If another incoming new term is like one already in core, the coefficients are added. If not, the new term is written out on drum, but no comparison with other terms on the drum is made. In that way we are sure that the terms in core and those on the drum form mutually exclusive sets.

The collection is maintained as a set of balanced binary trees. Each node of the trees contains a term of the expression and may contain left and right links to subtrees. All terms are in a canonical form and they are ordered within the trees. Scanning the collection to find a given term is then done using a binary search. Because we are interested in cases where the number of different terms is much less than the total number of searches, we need not worry about the time required to build and balance the trees.

Once all terms to be produced by the current control stack of commands have either been added to the collection or saved on drum, the collection in core is linearized and written to another part of the drum. Then a new control stack is created which takes the uncollected terms on drum and tries once more to collect them. This pass begins with an empty collection in core. Once more, if the collection is finally closed, overflow terms are written back on the drum. This process continues until there is no more overflow and the final collection has been written on the drum. At this time the overflow area is empty and a new control stack can be created in response to user commands. This control stack will take terms from the collections on the drum. Since we only want one term at a time and we do not care about their order, the linear storage of the drum is quite satisfactory. You do have to pay the price of multiple attempts to collect some terms. Still greater speed could probably be attained by using a hash-search and by double buffering the drum I/O. However, we are now able to do the collection of like terms on expressions with many hundreds of thousands of terms before collecting and many tens of thousands of terms after collection. By simply setting some operating modes, a user can substitute tapes for the drum areas. This effectively removes any storage limitation on the size of expressions.

One of the noteworthy special features which we developed for our problem is the integration package.[4] It expects input, in the form of transcendentals times rationals, in fully partial-fractioned form. It must be given a list of possible linear or quadratic factors which may appear in denominators and a list of transcendentals, their derivatives and a few terms of their expansions around the end points of the integration region (if the rational factors can become singular there). The main technique is integration by parts. If the rational factor is not a single denominator factor, it integrates by parts and for our class of functions this reduces the degree of the transcendentals by 1. If this introduces new denominators the result is first partial fractioned and then sent to the integrator again. Integrands with single denominator factors are done by explicit substitution. These

generally increase the degree of the transcendental. This simple technique is very powerful. In our case, our final integral consists of trilogarithms, dilogarithms, logarithms, polynomials and rational factors of the form $1/x$, $1/(1-x)$, $1/(1+x)$, $1/(1-x+x^2)$ to be integrated from 0 to 1. Our program evaluates integrals consisting of 1000 such terms in about 2 minutes. Besides the virtue of speed, this program is easily extended to include more transcendentals or rational functions, and the new additions are easily checked numerically.

In the realm of conjecture parallel processing is a very promising approach for algebraic work organized in this way. In particular the multi-mini processor configuration with many small, simple processors sharing a core area should greatly speed up execution. Because very little use is made of non-indexing arithmetic operations, the lack of fast floating point and double precision arithmetic operations is not important.

The 'one at a time' processing scheme allows many processors to share a common control stack, but each one could completely process a different term of the input expression, including threading its separate way down shared trees in collecting terms. The amounts of 'impure storage' generated while processing terms is generally negligible, so that multiplying this by the number of processors puts no strain on the system. When finished, the processor could return to pick up another term of the input expression.

We feel that no other existing algebraic program meets the requirements for our calculation. Others may be too inefficient in time or storage management, or they may be too inaccessible or too inflexible. We hope to have convinced you that ASHMEDAI is a very useful program.

But it can be improved. The program has been developed and used by physicists. For us physicists, the problems which the system can attack are more important than the system itself. We have often added new features to get us through a particular bottleneck. After that problem is solved, instead of going back to polish the new feature, we press on to another calculation. As fast as the program may be, no effort has been put into general timing checks or into algorithm optimization. We have used whatever seemed to be a good idea unless forced by the problem to do better. So the program might benefit from some face-lifting. For example, since the program was designed primarily for batch, there is no recovery from syntax errors in the interactive mode although this is easy to implement. The user interface can be markedly improved. The user's guide needs to be more systematic. Certain commonly used constructions ought to have simple mnemonics to call them. Some awkward user commands should be streamlined. This is like a traditional packaging problem in marketing. The contents are worthwhile, but what meets the consumer's eye is very important. We do not have the time or the training for this job. Perhaps one of you will take up the challenge. The result would be a powerful system which was also more attractive and easy to use.

VERIFICATION

In actually working on a difficult problem such as the anomalous magnetic moment of the electron, much more time is spent verifying the work than is spent performing it or creating the algebraic system. The situation is analogous to experimental work. Most experimentalists realize this difficulty and describe their experimental crosschecks in great detail. Such is often not the case with people new to algebraic work. For this reason we wish to discuss both the possible sources of error and the steps we take to avoid them. Rest assured that these errors do arise however improbable some may seem.

The first category is that of machine faults. While computers are quite reliable, any calculation requiring 10^{12} machine cycles is in peril. In a Monte Carlo integration, most errors occurring once in 10^{12} operations are statistically insignificant. That is not the case in this work. More likely, however, are errors introduced by peripheral equipment: card readers and punches and tape, drum and disk drives. Most such errors are detected by the equipment. Some are not detected by the equipment and are not immediately fatal.

The second category is that of program (ASHMEDAI) bugs. These can be due to the proper coding of an improper algorithm or due to coding errors. It must always be born in mind that no very large program can ever be assumed to be fully debugged.

Finally, there are errors committed in the use of such a system. The entire idea behind a certain approach may be fundamentally in error or a correct approach can be badly implemented either in some general cases or in very obscure limiting cases. The sequences of commands can be at fault or the fault can be in erroneous substitutions or in incorrect integral tables. These in turn can be the results of mistakes in hand work or simply of badly punched cards.

The verification procedures we employ are elaborate but probably not foolproof. Simple hardware errors are kept in check by keeping track of the general error rate. Since we constitute a small fraction of the total machine load, we are not particularly vulnerable. Punch and reader errors are largely fatal and cause the system to stop executing. In addition, we sequence all ASHMEDAI output cards and test the sequencing on input. Input syntax is heavily constrained. Although symbols are not explicitly declared, we can look at the symbol table to detect senseless symbols created through errors.

Coding errors in ASHMEDAI are a constant source of worry. This is heightened by frequent changes in the program. The data structures have a degree of redundancy built in as a check. At a very small expense in time this redundancy is exploited to detect gross program errors. In addition, after changes to the program we test both on the simple examples contained in the tutorial[5] and on sample parts of major calculations. Coding errors probably are not algebraically consistent. Doing the same problem a variety of ways exploits this feature to detect errors.

Since most of the processing involves performing replacements specified in identities, checking these identities is a major concern. These identities can be broken into two broad groups. The first is those that are numerically consistent: they are simple numerical identities if the indeterminates are given appropriate numerical values. An example is partial fraction decomposition.

The second category which includes integration, differentiation and limiting replacements do not make numerical sense pointwise.

The numerically consistent identities are easiest to verify. We have developed a separate program which reads the actual cards used by the ASHMEDAI program. The left and right hand sides of identities are evaluated numerically and compared. Where necessary we precede a set of identities by additional identities which give numerical values to the basic indeterminates and then define the secondary indeterminates in terms of the basic ones (e.g. X=1.23 $ XP = 1+X $ IXM = 1/(1-X)). This technique has completely supplanted hand checking in our work where it is applicable.

Identities which are not numerically consistent are more difficult to verify. The largest subgroup in this class is the basic integral tables. Our mainstay here is to compare an evaluation of the actual ASHMEDAI data cards containing the integral with a precise numerical integration of the integrand. Integrals which are divergent or which otherwise depend on a parameter taken to a limiting value are evaluated either for finite values of the parameters or are modified so as to extract the dependence upon the limit. Those functions which contain reference frame independent 4-vector quantities are evaluated in a specific reference frame. Even in cases where we do algebraic work because the corresponding numerical work would not have sufficient accuracy, it is possible to test the basic integrals in the tables to high precision. We consider it unlikely that an analytic error at this level leads to extremely small numerical errors. The differentiation identities and other parts of the program doing integration by parts can also be verified using particular sample cases.

Just as constraints on the syntax of input data structures help find errors, so too constraints on the syntax of the command language make most command errors fatal. Sequences of commands which would contradict the basic 'control stack' architecture and improper references to data structures are detected.

Beyond all these details we rely upon the physical sense of the results and their comparison with results obtained by vastly different methods and by other research groups. QED is sullied by certain well-understood singularities in the theory. In addition, our techniques introduce non-existent divergences into partial results. A delicate cancellation between terms yields a finite answer. Errors which do not preserve the algebraic sense of the expression destroy this cancellation and thereby reveal themselves. The previous numerical calculations of contributions to the electron's magnetic moment in this order of perturbation theory (good to a few percent) serve also as a constraint upon the analytic results. Finally, it is important that the results be obtained quite independently by another group using different machines, programs and techniques. This, in the final analysis, is the only truly convincing verification.

CONCLUSIONS

We would like to reiterate several conclusions we have drawn from this work. Time limitations cannot be ignored in many physically interesting problems. Our experience indicates that investing symbols with their full algebraic significance can increase execution time prohibitively. We have encountered this difficulty in the areas of rational function algebra and of symbolic integration. Because of this, we find it useful to distinguish between symbolic manipulation and symbolic algebra. Our substitutions manipulate symbols based on the forms contained in the identity statements without recognition of the 'underlying' algebraic meaning of the indeterminates. In many applications, this approach is what allows us to succeed.

Storage space limitations are also crucial. We succeed only by using auxiliary storage. Since this storage is not randomly accessible at high speed we must make accommodations in our internal data structures. While the complexity of algebraic expressions leads one into highly interconnected structures, we feel that a readily segmentable structure is absolutely vital.

Finally, we wish to emphasize that people at other institutions use this system on a variety of problems. The package is not limited to our problem or even other QED problems. Its usefulness comes from being written in a common computing language, from its efficient use of storage and rapid execution time.

REFERENCES

1. For details about ASHMEDAI, see M.J. Levine, USAEC Report No. CAR-882-25, 1971) (unpublished), Available from Physics Dept. Carnegie-Mellon Univ., Pittsburgh, Pa.; M.J. Levine, Proceedings of the Third International Colloquium in Advanced Computing Methods in Theoretical Physics, edited by A. Visconti, Marseille, 1973 (unpublished).
2. For details see R. Roskies, Computational Aspects of Quantum Electrodynamics: The Lepton g Factors, published in Particles and Fields - 1974, AIP Conference Proceedings No. 23, edited by C.A.Carlson, New York, 1975.
3. M. J. Levine and R. Roskies, submitted to Phys. Rev. D, Comments and Addenda.
4. For details, see M.J.Levine and R.C. Perisho, USAEC Report No. COO-3066-37, 1974, (unpublished). Available from Physics Dept., Carnegie-Mellon University, Pittsburgh, Pa.
5. See R. C. Perisho, ASHMEDAI User's Guide, USAEC Report No. COO-3066-44, 1975 (unpublished); Available from Physics Dept., Carnegie-Mellon Univ., Pittsburgh, Pa.,15213.

*Work supported in part by the U.S.Atomic Energy Research and Development Administration under Contract No. E(11-1)-3066.

†Alfred P. Sloan Fellow. Work supported in part by the National Science Foundation under Contract No.MPS 75-14073.

An Approach to Automatic Asymptotic Expansions

Richard J. Fateman
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, California 94720

Abstract

An example of a multiple-scale perturbation expansion using the MACSYMA algebraic manipulation system is derived. The problem solved is a differential equation for a linear damped oscillator, whose closed form solution can be easily compared to the generated approximation. The advantages of the automatic approach include, especially in more difficult cases, the rapid computation of higher order terms by computer, complete accuracy in algebra, and automatic and accurate conversion of the formulas to graphical or numerical form. Among the unusual features used is MACSYMA's ability to solve in closed form, elementary ordinary differential equations.

I. Introduction

Most mathematical problems facing applied mathematicians cannot be solved exactly and analytically. Principal among the approximation tools provided by algebra systems has been expansion by Taylor series. For example, CAMAL, ALTRAN, MACSYMA and numerous other programs provide for the automatic expansion of functions and/or the automatic handling of truncated power series arithmetic. While solution of differential equations (the usual application) by Taylor series sometimes generates as subproblems the need to solve systems of linear equations with symbolic coefficients, and occasionally systems of algebraic equations, these tools are not sufficient for the application here.

In fact, the facilities used here include MACSYMA's ability to solve, in closed form, not only algebraic systems, but ordinary differential equations. This is a particularly interesting point to us, because in and of itself, such a facility might be of little interest. To be more precise, any applied mathematician worth his salt might likely scoff at a program which could solve only the very standard cookbook ordinary differential equations (ODEs) that sophomores in college are taught to solve. The MACSYMA ODE solver, modelled in part along the thesis work of Joel Moses, and certainly heavily reliant on the advanced integration techniques in MACSYMA, has lain fallow for a number of years precisely because it was typical of capabilities for which the need had not yet arisen. Recent improvements have made the facility even more useful in the application described here, but to our knowledge, this is the first application of this program.

(An implementation note: The programs are written using MACSYMA's command-level language and pattern matcher, not in LISP.)

Aside from Taylor series, there are a few systems which allow for a Fourier-type expansion "Poisson Series". Among these systems we include MACSYMA, CAMAL, and systems specialized for celestial mechanics calculations. (See the Barton-Fitch survey [Barton, Fitch, 72] for a discussion.)

The technique we describe here is usually called the method of multiple scales, although minor variations or specific applications are sometimes given other names [Nayfeh, 73]. The vast literature on perturbation methods and asymptotic analysis has only recently become somewhat organized [Olver, 74], but it is clear that theory still lags behind practice in this discipline.

Without dwelling on details for the moment, the intent of this method is to approximate a solution $x(t,\varepsilon)$ of a differential equation, where ε is a small (constant) perturbation. The approximations to $x(t,\varepsilon)$ should approach the solution to the unperturbed equation as $\varepsilon \to 0$ for all t. As we will show by example in the next section, Taylor series in ε typically fail to satisfy this requirement. Further justification for this approach and many references may be found in [Nayfeh, 73], where the specific example we deal with is presented in section 6.1. Other papers of interest describing the similar computational assistance of symbol manipulation programs are [Cohen, Bark, 76], [Atherton, Homsy, 73] and [Barthès-Biesel, Acrivos, 73].

II. An Example

Consider the linear damped oscillator

$$\ddot{x} + x = -2\varepsilon\dot{x} \; . \qquad (1)$$

This happens to have the closed form solution (easily obtained in MACSYMA)

$$x = e^{-\varepsilon t}\{k1 \sin(\sqrt{1-\varepsilon^2}\,t) + k2 \cos(\sqrt{1-\varepsilon^2}\,t)\} \; . \qquad (2)$$

Actually, MACSYMA's result is not quite so well simplified, the argument of the sin and cos being

$\sqrt{4-4\varepsilon^2}\,t/2$, but this does not interfere with the usefulness of the answer.

In the absence of a closed form solution, one approach to an analytical approximation is to compute a Taylor series expansion in ε (presumably a small parameter) for x. Using MACSYMA we can easily obtain

$$\begin{aligned} x = {} & k1\,\sin(t) - k2\,\cos(t) \\ & + (k2\,\cos(t) - k1\,\sin(t))t\varepsilon \\ & - (k2\,\cos(t) - k1\,\sin(t))t^2 \\ & + (k2\,\sin(t) + k1\,\cos(t))t\varepsilon^2/2 \\ & + \cdots \end{aligned} \qquad (3)$$

where the "..." signifies terms of order $\varepsilon^3 t^3$. As $t \to \infty$, the exact solution (2) approaches 0 as an exponential with superimposed oscillations, but the Taylor series (3) truncated at any finite power n of ε fails to accurately reflect the solution since the error grows as $\varepsilon^n t^n$. Nevertheless, an approximate solution to (1) can be mechanically generated of the form

$$\begin{aligned} x = {} & e^{-\varepsilon t}(k1\,\sin(t - \tfrac{1}{2}\varepsilon^2 t) + k2\,\cos(t - \tfrac{1}{2}\varepsilon^2 t)) \\ & + O(\varepsilon^2) \end{aligned} \qquad (4)$$

which accurately represents the asymptotic behavior of the solution, as $t \to \infty$, the error being of the form $O(\varepsilon^n)$ rather than $O(\varepsilon^n t^n)$.

Such approximations are naturally of some interest, and many techniques have been discovered to deduce these asymptotic expansions. One of these, the method of multiple scales, is illustrated by means of this example, and the specific manipulations required are performed through commands to the MACSYMA algebraic manipulation system.

III. The Method of Multiple Scales

The principal behind the method of multiple scales is that by imposing a sequence of time scales, the solution for all time will be successively more closely approximated.

Let the time scales be

$$T_0 = t,\ T_1 = \varepsilon T_0, \ldots, T_m = \varepsilon^m T_0\ , \qquad (5)$$

and let the solution be expressed as

$$\begin{aligned} x(t) &= X(T_0, T_1, \ldots, T_m; \varepsilon) \\ &= x_0(T_0, \ldots, T_m) + \varepsilon x_1(T_0, \ldots, T_m) + \cdots \\ &\quad + \varepsilon^{m-1} x_{m-1}(T_0, \ldots, T_m) + O(\varepsilon T_m)\ . \end{aligned} \qquad (6)$$

The error is stated as $O(\varepsilon T_m)$ to remind the reader that this expansion is valid for times up to $O(\varepsilon^{-m})$, but beyond these times the expansion should be continued to more terms.

We have transformed an ordinary differential equation into a partial differential equation. More generally, a PDE will be replaced by one with more independent variables.

It is not clear that we can compute the sequence of functions $x_0, x_1, \ldots, x_m$, and in fact, additional conditions must be imposed. If equation (6) is to be valid for times as large as ε^{-m}, then $\varepsilon^m x_m$ must be a small correction to $\varepsilon^{m-1} x_{m-1}$ which must be a small correction to $\varepsilon^{m-2} x_{m-2}, \ldots$. Thus we insist that x_i / x_{i-1} is bounded for all t and $i = 1, \ldots, m$.

By solving a sequence of ODEs for the x_i, and carrying through some tedious arithmetic, we can obtain a solution which is uniformly valid. (Roughly speaking, the error incurred by truncating the process after m terms is small regardless of t).

The technique given becomes uncomfortably laborious, in most cases, when m = 1 or 2. Assuming the associated problems can be solved (the generated ODEs and the "secularity" condition that $x_i / x_{i-1} < \infty$), the methods used can be extended to higher orders automatically. It is apparent that in most textbook examples only one or two terms are generated; it is not clear whether this is because more terms are uninteresting, or just too difficult to obtain.

The method in gross outline consists of substituting equations (5) and (6) into equation (1); then, by equating coefficients of like powers of ε to zero, we obtain a sequence of ODEs for $x_0, \ldots, x_m$. These equations are solved successively, each time removing the "secular" terms which grow too rapidly for x_i / x_{i-1} to be bounded. Details are given in the next section.

IV. A Dialogue with MACSYMA

Let us assume, for this example, that expansion with respect to three time scales will suffice. That is, the solution x will be a function of t_0, t_1 and t_2. In order to inform MACSYMA that derivatives of x with respect to these variables should be retained symbolically, we type the command

```
(C1) DEPENDENCIES(X(T0,T1,T2))$
```

The gradients of t_0, t_1 and t_2 with respect to t implied by (6) are defined by use of the GRADEF command. For brevity we type the commands as a group, enclosed by parentheses, separated by commas:

```
(C2) (GRADEF(T0,T,1),GRADEF(T1,T,E),
      GRADEF(T2,T,E↑2))$
```

We know that when we substitute equation (6) into equation (1), powers of ε (denoted E) will be generated which are higher than we care to keep around. There are several ways of automatically converting powers of ε exceeding 2 to zero in MACSYMA expressions. For example, the rational function truncation routines (RATWEIGHT and RATWTLVL settings) could be used. We demonstrate a method which does not insist on forcing expressions into a polynomial canonical form. So long as we do not divide by ε, we can alter MACSYMA's simplifier to convert $\varepsilon \uparrow n$ for $n > 2$ to zero:

```
(C3) NNPRED(N) := IS(N>2)$
(C4) MATCHDECLARE(NN,NNPRED)$
(C5) TELLSIMPAFTER(E↑NN,0);
(D5) [**RULE1,SIMPEXPT]
```

The last command, terminated by a ";" rather than a "$" displays its answer, in this case a list including a name by which we can later refer to this rule.

We wish to distribute multiplication over addition at least enough to collect powers of ε, so we command:

```
(C6) EXPOP:3$
```

A sample solution is $x_0 + \varepsilon x_1 + \varepsilon^2 x_2$, which we will call H:

```
(C7) H: X[0]+E*X[1]+E↑2*X[2]$
```

We substitute this value for H into equation (1):

```
(C8) EQ:DIFF(H,T,2) + H + 2*E*DIFF(H,T);
```

$$(D8)\ \left(\frac{D^2}{DT0^2}\ X_2\right)E^2 + 2\left(\frac{D^2}{DT0\ DT1}\ X_1\right)E^2 + 2\left(\frac{D}{DT0}\ X_1\right)E^2 + \left(\frac{D^2}{DT1^2}\ X_0\right)E^2 + 2\left(\frac{D}{DT1}\ X_0\right)E^2 + 2\left(\frac{D^2}{DT0\ DT2}\ X_0\right)E^2 + X_2E^2 + \left(\frac{D^2}{DT0^2}\ X_1\right)E + 2\left(\frac{D^2}{DT0\ DT1}\ X_0\right)E + 2\left(\frac{D}{DT0}\ X_0\right)E + X_1E + \frac{D^2}{DT0^2}\ X_0 + X_0$$

Now let us undo some of the earlier commands in order to speed up processing:

```
(C9) (EXPOP:0,KILL("**RULE1"))$
```

To isolate equations for x_0, x_1 and x_2, we pick out the various coefficients of powers of E in EQ:

```
(C10) COEF[0]:SUBST(0,E,EQ);
```

$$(D10)\ \frac{D^2}{DT0^2}\ X_0 + X_0$$

```
(C11) COEF[1]:COEFF(EQ,E,1);
```

$$(D11)\ \frac{D^2}{DT0^2}\ X_1 + 2\left(\frac{D^2}{DT0\ DT1}\ X_0\right) + 2\left(\frac{D}{DT0}\ X_0\right) + X_1$$

```
(C12) COEF[2]:COEFF(EQ,E,2);
```

$$(D12)\ \frac{D^2}{DT0^2}\ X_2 + 2\left(\frac{D^2}{DT0\ DT1}\ X_1\right) + 2\left(\frac{D}{DT0}\ X_1\right) + \frac{D^2}{DT1^2}\ X_0 + 2\left(\frac{D}{DT1}\ X_0\right) + 2\left(\frac{D^2}{DT0\ DT2}\ X_0\right) + X_2$$

We need to solve COEF[0], an ordinary differential equation. While this can be done by inspection, requiring this in general would prohibit successful automation of the calculation.

A program (SOLDIER) written by Joel Moses and partially described in [3] demonstrated the feasibility of solving simple ODEs. Another program, implemented in MACSYMA (ODE2) by Ben Kuipers and improved by Jeffrey Golden, has a similar capability, and is in some ways superior to Moses' earlier effort: ODE2 and its subroutines are all written entirely in MACSYMA's Algol-like programming language. The program is only about 6 pages long and can be easily understood and modified. The program draws heavily on the power of MACSYMA's integration program, a distinct advantage.

The programs were automatically translated into LISP for the sake of compactness; while their speed may have improved slightly, this was not an issue, since most of the time is spent inside compiled subroutines.

These programs are automatically loaded into core memory when they are called by name (ODE2 in this case).

One of the (many) MACSYMA system flags controls the dominant mode for expressing trigonometric functions. While it is not essential, the solutions to these differential equations are more compact when expressed in terms of complex exponentials. Thus:

```
(C13) EXPONENTIALIZE:TRUE$
```

Now we proceed to call ODE2 to find x_0 as a function of t_0:

```
(C14) SOL[0]: ODE2(COEF[0],X[0],T0);
```

$$(D14)\ X_0 = K1\ \%E^{\%I\ T0} + K2\ \%E^{-\%I\ T0}$$

Note that %E is the notation for the base of the natural logarithms and $\%I = \sqrt{-1}$.

We must declare the "constants" K1 and K2 to be functions of T1 and T2, and furthermore, we had better change their names (say to K01 and K02) because the next time we call ODE2, the two (probably unrelated) constants will still be called K1 and K2. We could build a "shell" around ODE2 to rename the constants each time, but this would add unnecessary material to this paper.

```
(C15) DEPENDENCIES(K01(T1,T2),K02(T1,T2))$
```

The following command sets X[0] to the result of evaluating the right-hand-side (RHS) of SOL[0] in a context where K1 is replaced by K01, K2 by K02:

```
(C16) X[0]:EV(RHS(SOL[0]),K1=K01,K2=K02);
```

$$(D16)\ K01\ \%E^{\%I\ T0} + K02\ \%E^{-\%I\ T0}$$

Now evaluate COEF[1], where X[0] now has a value, and perform the differentiations:

```
(C17) EV(COEF[1],DIFF);
```

(D17) $2(\%I\ \frac{DK01}{DT1}\ \%E^{\%I\ T0} - \%I\ \frac{DK02}{DT1}\ \%E^{-\%I\ T0})$

$+ 2(\%I\ K01\ \%E^{\%I\ T0} - \%I\ K02\ \%E^{-\%I\ T0})$

$+ \frac{D^2}{DT0^2}\ X_1 + X_1$

Now solve this last equation (which can be referred to by "%") for X[1] as a function of T0: (Note, D18 is an expression, not an equation. MACSYMA assumes, in such cases, that the right hand side is zero.)

(C18) SOL[1]: ODE2(%,X[1],T0);

(D18) $X_1 = -\%E^{-\%I\ T0}(T0(2\ \frac{DK01}{DT1}\ \%E^{2\ \%I\ T0}$

$+ 2\ K01\ \%E^{2\ \%I\ T0} + 2\ \frac{DK02}{DT1} + 2\ K02)$

$+ \%I\ \frac{DK01}{DT1}\ \%E^{2\ \%I\ T0}$

$+ \%I\ K01\ \%E^{2\ \%I\ T0} - \%I\ \frac{DK02}{DT1}$

$- \%I\ K02)/2 + K1\ \%E^{\%I\ T0}$

$+ K2\ \%E^{-\%I\ T0}$

We must apply the secularity condition, which is the only non-routine step in this whole procedure. For x_1/x_0 to be bounded for all $t = t_0$, we note that $x_1 = \alpha t_0 + \beta$, α and β being oscillatory functions, and if x_0 is bounded and α is zero, the secularity condition appears to be satisfied. The following steps pick out the coefficient of t_0, then pick out the essential equations required to make the coefficient become zero:

(C19) RATCOEF(RHS(%),T0);

(D19) $-\%E^{-\%I\ T0}((\frac{DK01}{DT1} + K01)\%E^{2\ \%I\ T0}$

$+ \frac{DK02}{DT1} + K02)$

(C20) EXPAND(%);

(D20) $-\frac{DK01}{DT1}\ \%E^{\%I\ T0} - K01\ \%E^{\%I\ T0}$

$- \frac{DK02}{DT1}\ \%E^{-\%I\ T0} - K02\ \%E^{-\%I\ T0}$

(C21) COEFF(%,%E↑(%I*T0));

(D21) $-\frac{DK01}{DT1} - K01$

(C22) ODE2(%,K01,T1);

(D22) $K01 = C\ \%E^{-T1}$

We see that K01(T1,T2) = K001(T2)*%E↑(-T1), and similarly for K02. Substituting these values back in the expressions for x_0 and x_1 we find

(C23) DEPENDENCIES(K001(T2),K002(T2))$

(C24) K01:K001*%E↑(-T1)$

(C25) K02:K002*%E↑(-T1)$

(C26) X[0]:EV(X[0]);

(D26) $K001\ \%E^{\%I\ T0-T1} + K002\ \%E^{-T1-\%I\ T0}$

(C27) X[1]:EV(RHS(D19),DIFF);

(D27) $K1\ \%E^{\%I\ T0} + K2\ \%E^{-\%I\ T0}$

Recall that ODE2 introduces K1 and K2, and we must change their names and declare dependencies as we did before.

(C28) DEPENDENCIES(K11(T1,T2),K12(T1,T2))$

(C29) X[1]:EV(X[1],K1=K11,K2=K12):

(D29) $K11\ \%E^{\%I\ T0} + K12\ \%E^{-\%I\ T0}$

It would seem to be the next order of business to solve for x_2, but in fact we can (and should, for the sake of brevity of calculation and presentation) figure out the dependence of K001 and K002 on t_2 and then proceed to work on the "constants" in the value for x_1, namely K11 and K12.

A differential equation for x_2 is derived from COEF[2] by evaluation and differentiation of the quantities previously computed:

(C30) EV(COEF[2],DIFF);

(D30) $2(\%I\ \frac{DK001}{DT2}\ \%E^{\%I\ T0-T1}$

$- \%I\ \frac{DK002}{DT2}\ \%E^{-T1-\%I\ T0})$

$+ 2(-K001\ \%E^{\%I\ T0-T1} - K002\ \%E^{-T1-\%I\ T0})$

$+ K001\ \%E^{\%I\ T0-T1} + K002\ \%E^{-T1-\%I\ T0}$

$+ 2(\%I\ \frac{DK11}{DT1}\ \%E^{\%I\ T0} - \%I\ \frac{DK12}{DT1}\ \%E^{-\%I\ T0})$

$+ 2(\%I\ K11\ \%E^{\%I\ T0} - \%I\ K12\ \%E^{-\%I\ T0})$

$+ \frac{D^2}{DT0^2}\ X_2 + X_2$

Before solving this, we can eliminate those terms in this equation which will produce secular terms in the answer. This is a fairly general trick, and requires that we pick out the terms contributing to the particular solution. This is done below, and the several equations resulting from imposing secularity constraints are solved. While not apparent from the computer algebra as displayed, the differential equation (D30) can be transformed to

$$\frac{\partial^2 x_2}{\partial t_0^2} + x_2 = Q_1 e^{it_0} + Q_2 e^{-it_0}$$

with a particular solution of the form

$$x_2 = \tfrac{1}{2}i(Q_1 t_0 e^{it_0} - Q_2 t_0 e^{-it_0})$$

resulting in the growth with respect to t_0 making x_2/x_1 unbounded. The secularity condition is implemented by insisting $Q_1 = Q_2 = 0$. (Programming notes: RATNUMER picks out the numerator from an expression in rational form, the form in which RATCOEF returns its answer. Rational forms, signified by a /R/ after the line label, are in an internal canonical form based on a ratio of polynomials.)

```
(C31) %-DIFF(X[2],T0,2)-X[2]$
(C32) RAT(D32,%E↑(%I*T0));
```

$$(D32)\ /R/\ (((2\ \%I\ \frac{DK11}{DT1} + 2\ \%I\ K11)\%E^{T1} + 2\ \%I\ \frac{DK001}{DT2} - K001)(\%E^{\%I\ T0})^2 + (-2\ \%I\ \frac{DK12}{DT1} - 2\ \%I\ K12)\%E^{T1} - 2\ \%I\ \frac{DK002}{DT2} - K002)/(\%E^{T1}\%E^{\%I\ T0})$$

```
(C33) RATCOEF(RATNUMER(%), %E↑(2*%I*T0));
```

$$(D33)\ /R/\ (2\ \%I\ \frac{DK11}{DT1} + 2\ \%I\ K11)\%E^{T1} + 2\ \%I\ \frac{DK001}{DT2} - K001$$

The expression labelled D33 is Q_1, and can be solved by K11 as a function of T1; Q_2, displayed on line D35 can be solved by K12 as a function of T1, and the appropriate new "constants" (in this case, functions of T2) inserted.

```
(C34) ODE2(D33,K11,T1);
```

$$(D34)\ K11 = (C - \frac{(2\ \frac{DK001}{DT2} + \%I\ K001)T1}{2})\%E^{-T1}$$

```
(C35) RATNUMER(D32) - D33*%E↑(2*%I*T0);
```

$$(D35)\ /R/\ (-2\ \%I\ \frac{DK12}{DT1} - 2\ \%I\ K12)\%E^{T1} - 2\ \%I\ \frac{DK002}{DT2} - K002$$

```
(C36) ODE2(D35,K12,T1);
```

$$(D36)\ K12 = (\frac{(\%I\ K002 - 2\ \frac{DK002}{DT2})T1}{2} + C)\%E^{-T1}$$

```
(C37) DEPENDENCIES(K011(T2),K012(T2))$
(C38) K11:SUBST(K011,C,RHS(D34));
```

$$(D38)\ (K011 - \frac{(2\ \frac{DK001}{DT2} + \%I\ K001)T1}{2})\%E^{-T1}$$

```
(C39) K12:SUBST(K012,C,RHS(D36));
```

$$(D39)\ (\frac{(\%I\ K002 - 2\ \frac{DK002}{DT2})T1}{2} + K012)\%E^{-T1}$$

Now if we put these values computed so far into x_1, we see that the ratio x_1/x_0 will remain bounded for times on the order of εt only if the coefficient of T_1 is zero. This secularity condition gives us a chance to remove the variation with respect to T_2 of the "constants" K001 and K002.

```
(C40) X[1],EVAL;
```

$$(D40)\ (K011 - \frac{(2\ \frac{DK001}{DT2} + \%I\ K001)T1}{2})\%E^{\%I\ T0-T1} + (\frac{(\%I\ K002 - 2\ \frac{DK002}{DT2})T1}{2} + K012)\%E^{-T1-\%I\ T0}$$

```
(C41) RATCOEF(D40,T1*%E↑(%I*T0-T1));
```

$$(D41)\ -\frac{2\ \frac{DK001}{DT2} + \%I\ K001}{2}$$

```
(C42) ODE2(D41,K001,T2);
```

$$(D42)\ K001 = C\ \%E^{-\frac{\%I\ T2}{2}}$$

```
(C43) K001:SUBST(K0001,C,RHS(%));
```

$$(D43)\ K0001\ \%E^{-\frac{\%I\ T2}{2}}$$

At the risk of being accused of omitting vital steps, we will shorten subsequent computational instructions, and assure the reader that in fact repeating this process of generating and solving ODE's presents no new problems. After a few steps we can express our result to first order:

```
(C56) X[0]+E*X[1],EV;
```

$$(D56)\ E(K0012\ \%E^{\frac{\%I\ T2}{2} - T1 - \%I\ T0} + K0011\ \%E^{-\frac{\%I\ T2}{2} - T1 + \%I\ T0}) + K0002\ \%E^{\frac{\%I\ T2}{2} - T1 - \%I\ T0} + K0001\ \%E^{-\frac{\%I\ T2}{2} - T1 + \%I\ T0}$$

Since we have chosen an example for which a closed form solution exists, we can in fact compare the result of this analysis to the true solution. Let us impose some initial conditions so that the exact answer as computed on line D5 can be converted to a more compact equivalent form as suggested by [Nayfeh 73]. We type this form in on line C18. To match D18 and the right hand side of D5 at T = 0 and their derivatives at T = 0, we find that suitable values for K1 and K2 are given on lines E25 and E26.

```
(C4) 'DIFF(X,T,2)+X=-2*E*'DIFF(X,T);
```

$$(D4)\ \frac{D^2X}{DT^2} + X = -2E\ \frac{DX}{DT}$$

```
(C5) ODE2(D4,X,T);
IS (E-1)(E+1) POSITIVE, NEGATIVE, OR ZERO?
NEGATIVE;
```

$$(D5)\ X = \%E^{-ET}(K1\ SIN(\frac{SQRT(4-4E^2)T}{2}) + K2\ COS(\frac{SQRT(4-4E^2)T}{2}))$$

...

```
(C18) A*%E↑(-E*T)*(COS(T*SQRT(1-E↑2)+PHI));
```

$$(D18)\ A\%E^{-ET}COS(SQRT(1-E^2)T+PHI)$$

```
(C19) DIFF(%)),T);
```

$$(D19)\ -A\ SQRT(1-E^2)\%E^{-ET}SIN(SQRT(1-E^2)T+PHI) - AE\%E^{-ET}COS(SQRT(1-E^2)T+PHI)$$

```
(C22) RHS(D5)=D18,T=0;
```

(D22) K2 = A COS(PHI)

(C23) DIFF(RHS(D5),T) = D19$

(C24) %,T=0;

$$(D24)\ \frac{SQRT(4-4E^2)K1}{2} - E\ K2 = -\ A\ SQRT(1-E^2)SIN(PHI) - A\ E\ COS(PHI)$$

(C25) SOLVE([D22,D24],[K1,K2]);

SOLUTION

(E25) K2 = A COS(PHI)

$$(E26)\ K1 = -\ \frac{2A\ SQRT(1-E^2)SIN(PHI)}{SQRT(4-4E^2)}$$

(D26) [[E25,E26]]

If we substitute $t_0 = 1$, $t_1 = \varepsilon t$, $t_2 = \varepsilon^2 t$, and regroup terms in (D56) we find our generated approximate solution is

$$e^{-\varepsilon t}[(\varepsilon k0012 + k002)e^{i\varepsilon^2 t/2 - \varepsilon t - it} + (\varepsilon k0011 + k001)e^{-i\varepsilon^2 t/2 - \varepsilon t + it}] .$$

If we let $2(\varepsilon k0012 + k002) = Ae^{-i\phi}$ and $2(\varepsilon k0011 + k001) = Ae^{i\phi}$, then our approximate solution becomes, by applying deMoivre's rule then substituting $1 - \cos^2(\phi)$ for $\sin^2(\phi)$,

...

$$(D31)\ A\ SIN(PHI)\%E^{-ET}SIN(\frac{E^2T}{2} - T) + A\ COS(PHI)\%E^{-ET}COS(\frac{E^2T}{2} - T)$$

Then contracting this sum of trig terms by a built-in simplification command:

(C34) TRIGREDUCE(%);

$$(D34)\ A\ \%E^{-ET}COS(\frac{E^2T}{2} - T - PHI)$$

yields a form very close to the exact solution (D18) calculated previously. In fact it may be shown via TRIGREDUCE and Taylor series expansion of the square root, that the difference between D34 and D18 is

$$-2ae^{-\varepsilon t}\sin[(\frac{1}{2}\sqrt{1-\varepsilon^2} + 1 - \frac{1}{2}\varepsilon^2)t + \phi] \cdot \sin[\frac{1}{2}\sqrt{1-\varepsilon^2} - 1 + \frac{1}{2}\varepsilon^2)t]$$

$$\approx -2ae^{-\varepsilon t}\sin[(\sqrt{1-\varepsilon^2} + 2 - \varepsilon^2)t/2 + \phi] \cdot \sin[(-\varepsilon^4/16 + \cdots)t]$$

$$= O(1)\cdot O(\varepsilon^4 t) .$$

Thus this expansion is "good" for times as long as $t = 1/\varepsilon^3$.

The reader may feel deceived by not seeing these last few assertions proved entirely via MACSYMA. In fact, MACSYMA did provide most of the manipulation, once the suggestion of the form (D34) of the answer was extracted from [Nayfeh 73]. The steps required, however, resulted in several pages of output, parts of which were trivial (Taylor expansion of $\sqrt{1-\varepsilon^2}$) and parts of which were painful efforts to rearrange terms.

Some such steps appear to be more easily done by hand, since the computer system conventions must be over-ridden in various ways to produce a result which does not satisfy some built-in notion of simple.

V. Prospects and Difficulties

First, the difficulties we have encountered in this work to date:

1. There is a clear bias in published work for the "clever" approaches, as opposed to uniform approaches in problem solving. That is, of the many ways to write down a solution, the one published will ordinarily be more carefully tailored for the typesetter than for the person designing a general procedure.

2. The need to pick out subexpressions based on a global analysis of an answer (e.g., find the terms which tend to infinity as $t \to \infty$) is not easy to automate. In this particular situation, coefficient-finding routines were adequate only when guided by a user. We expect that criteria for removing secular terms can be more uniformly formulated.

3. Methods we hope to implement cannot provide asymptotic expansions with the same confidence we can provide, say, a polynomial factorization: The answer to the latter can be easily checked, and is computed entirely algorithmically. Some methods "work" yet are formally invalid. Furthermore, the asymptotic expansion may not be directly susceptible to any check, and may have been generated in a formally correct but unmanageable form. Using sines and cosines rather than complex exponentials in our example would lead to considerably larger expressions; on the other hand, recognizing that the two arbitrary constants in the solution of the harmonic oscillator can be represented as K1 = a + bi and K2 = complex conjugate of K1 leads to considerably smaller expressions.

4. Solutions to differential equations embody a wide class of functions for which effective arithmetic and simplification routines are not usually available. This is a strong contrast to polynomial or rational function problems. While there are some indications that progress can be made in these issues, there are more indications that truly effective (namely efficient, canonical) forms of practical utility are very hard to find. Principal among the optimistic signs is Moses' "General theory of special functions" [Moses 72] and recent progress in automating elliptic integration solutions [Ng 74]; among the pessimistic signs are the vast hodge-podge of information outside any currently available framework which seems accessible and even vital to the work of the best applied mathematicians [Olver 74].

Now for the prospects of improved facilities:

1. There appears to be a wide range of applicability for the method of multiple scales presented here. [Nayfeh 73] indicates some of the applications. Nevertheless, it is unreasonable to expect that the exact same steps with the substitution of another equation would produce a solution. Our hope is that a detailed annotated dialogue would serve as an adjunct to a text and as an implementation guide for the solution of similar problems. Additional dialogues following standard texts would further aid mathematicians, in the choice of a method and in the algebraic implementation of the method.

2. It is easier to consider an automated version of such a dialogue which could be placed in a library of methods. The user would have to apply less cleverness and/or intuition to solving the problem. We do not wish to minimize the effort in mathematical analysis and programming prowess necessary for this task; yet if texts in numerical analysis can be written with a heavy orientation toward FORTRAN programming examples, perhaps a similar approach to mechanization of the analytical methods of applied mathematics would be fruitful.

3. Given more information relating the mathematical problem to a physical context, a program may be able to choose one of the known methods as being most applicable. The simplest approach to this may be the development of problem-oriented program libraries and command-systems (or language modifications) based on a class of users and their problems. Pursuit of these matters depends on the development of coherent communities based on common problem areas in algebraic computation, and on software efforts analogous to those now being undertaken for numerical scientific programming support. We believe, with the decreased cost and increased accessibility of algebraic manipulation programs, it is only a matter of time before fruitful ventures of this nature become common.

Acknowledgments

This work was supported in part by the Energy Research and Development Administration through the Lawrence Livermore Laboratory of the University of California, Controlled Thermonuclear Research Computer Center, and by the Department of Defense Advanced Research Projects Agency, work order 2095 under Office of Naval Research Contract N00014-75-C-0661. The latter contract supports the MACSYMA computer system.

References

[Atherton, Homsy 73]
R.W. Atherton and G.M. Homsy, "Use of Symbolic Computation to Generate Evolution Equations and Asymptotic Solutions to Elliptic Equations," J. Comp. Physics 13 (1973) 45-58.

[Barthès-Biesel, Acrivos 73]
Barthès-Biesel, D. and A. Acrivos, "On Computer Generated Analytic Solutions to the Equations of Fluid Mechanics. The Case of Creeping Flows," J. Comp. Physics 12 (1973) 403-411.

[Barton, Fitch 72]
Barton, D. and Fitch, J.P., "Applications of Algebraic Manipulative Programs in Physics," Reports on Progress in Physics 35 (1972) 235-314. A shorter version appears as "A Review of Algebraic Manipulative Programs and Their Application," Computer J. 15 (Nov. 1972) 362-381.

[Cohen, Bark 76]
Cohen, I. and F. Bark, "Perturbation Calculations for the Spin Up Problem Using Reduce," TRITA-MEK-76-03, The Royal Institute of Technology, Department of Mechanics, S-10044 Stockholm 70, Sweden, March 1976.

[Mathlab 75]
Mathlab Group, The MACSYMA Reference Manual, version 8, The Laboratory for Computer Science, M.I.T., Cambridge, Mass. (1975).

[Moses 71]
Moses, J., "Symbolic Integration: The Stormy Decade," CACM 14 (Aug. 1971) 527-537.

[Moses 72]
Moses, J., "Toward a General Theory of Special Functions," CACM 15 (July 1972) 550-554.

[Nayfeh 73]
Nayfeh, Ali Hasan, Perturbation Methods, John Wiley, New York (1973).

[Ng 74]
Ng, Edward, "Symbolic Integration of a Class of Algebraic Functions," Proc. of EUROSAM 74, SIGSAM Bulletion 8 (Aug. 1974) 99-103.

[Olver 74]
Olver, F.W.J., Asymptotics and Special Functions, Academic Press, New York (1974).

A Study of Alternative Methods for the Symbolic Calculation of Elliptic Integrals

Edward W. Ng
Jet Propulsion Laboratory
California Institute of Tech.

Desanka Polajnar
Dept. of Computer Science
University of Southern Calif.

Abstract

This presentation describes the study of several candidate methods for the symbolic integration of a class of irrational expressions $R(x,y)$, where y^2 is a polynomial in x and R is a rational function of x and y. Three of these methods came from classical investigations of Legendre, Jacobi, and Weierstrass whereas one came from recent investigations of Carlson. The present paper may be considered a progress report on preliminary algorithmic study and some efforts of implementation on MACSYMA.

In this presentation, we describe the study of several candidate methods for the symbolic integration of a class of irrational expressions $R(x,y)$, where y^2 is a polynomial in x and R is a rational function of x and y. We shall also summarize some efforts of implementation on MACSYMA.

In a previous report (Ref. 2), we described an algorithmic scheme based on Carlson's R-functions of the form $R(a;b_1,\ldots,b_k;z_1,\ldots,z_k)$ to calculate elliptic integrals. The scheme is straightforward and mechanical in nature and appeared very appealling as a candidate for symbolic integration. However, as we launched into the implementation of the algorithm, there arose practical difficulties of two kinds. First, since the R-functions are multiparametered, it is difficult to find any straight forward algorithm to reduce a given R-function into canonical form, because reduction of a given parameter b_j may increase another parameter b_i. **Second**, there is no simple way for the intermediate simplification of any expression swell, which may make the entire reduction procedure intractable. For this reason we have re-examined some classical methods to determine if some parts can be combined with Carlson's analysis to yield an optimal algorithm in terms of both efficiency and storage. It is easy to reduce $R(x,y)$ to the form

$$R(x,y) = R_1(x) + P(x)/[Q(x)y] \qquad (1)$$

where R_1 is a rational function and P and Q are polynomials in x (Ref. 2). The integration of rational functions, in this case R_1, has been the subject of numerous investigations (cf. Ref. 1) and will not concern us here. In any final implementation, we shall assume that such capability is available on the particular system in question. Thus we turn our attention to the term $P/(Qy)$ in eq. (1). One promising candidate in the initial reduction of the expression $P/(Qy)$ is to apply a partial fraction decomposition to the rational function P/Q. First write

$$\int P/(Qy)\,dx = \int P_1/y\,dx + \int P_2/(Qy)\,dx \qquad (2)$$

If the degree of P is less than Q, P_1 will be zero. Otherwise the first integral in (2) is made up of linear combination of of integrals I_k of the form

$$I_k = \int x^k/y\,dx \qquad (3)$$

The rational function P_2/Q may be decomposed into partial fractions thereby reducing the second integral in eq. (2) to the form $\Sigma A_k H_k$, where

$$H_k = \int \frac{dx}{(x - \gamma_k)^k y} \qquad (4)$$

In general I_k and H_k satisfy recurrence formulae that can be derived from integration by parts. In particular, for the cases where y^2 is a third or fourth degree polynomial, these formulae will be inhomogeneous relations of three or four terms. Some examples may be cited from (Ref. 3). Let $y^2 = a_0 + a_1x + a_2x^2 + a_3x^3$, we have

$$(2k-1)a_3I_k = 2x^{k-2}y - 3(2k-2)a_2I_{k-1}$$
$$-3(2k-3)a_1I_{k-2} - (2k-4)a_0I_{k-3} \qquad (5)$$

$$2(k-1)b_3H_k = -\frac{2y}{(x-\gamma_k)^{k-1}} - 3(2k-3)b_2H_{k-1}$$

$$- 3(2k-4)b_1H_{k-2} - (2k-5)b_0H_{k-3},$$

$$b_3 \neq 0 , \quad k \neq 1 \qquad (6)$$

where $b_0 = a_3$, $b_1 = a_3\gamma_k + a_2$.

$$b_2 = a_3\gamma_k^2 + 2a_2\gamma_k + a_1 ,$$

$$b_3 = a_3\gamma_k^3 + 3a_2\gamma_k^2 + 3a_1\gamma_k + a_0$$

where $y^2 = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4$, we have,

$$(k-1)a_4I_k = x^{k-3}y - 2(2k-3)a_3I_{k-1}$$

$$- 6(k-2)\, a_2I_{k-2} - 2(2k-5)a_1I_{k-3} \qquad (7)$$

$$- (k-3)\, a_0I_{k-4} , \quad k \neq 1$$

$$(k-1)b_4H_k = -\frac{y}{(x-\gamma_k)^{k-1}} - 2(2k-3)b_3H_{k-1}$$

$$- 6(k-2)b_2H_{k-2} - 2(2k-5)b_1H_{k-3}$$

$$- (k-3)b_0H_{k-4}, \quad k \neq 1, \quad b_4 \neq 0 \qquad (8)$$

where $b_0 = a_4$, $b_1 = a_4\gamma_k + a_3$

$$b_2 = a_4\gamma_k^2 + 2a_3\gamma_k + a_2 ,$$

$$b_3 = a_4\gamma_k^3 + 3a_3\gamma_k^2 + 3a_2\gamma_k + a_1 ,$$

$$b_4 = a_4\gamma_k^4 + 4a_3\gamma_k^3 + 6a_2\gamma_k^2 +$$

$$+ 4a_1\gamma_k + a_0$$

These recurrences may be applied in principle to reduce the I_k's and H_k's to three basis functions, say, I_0, I_1, and H_1; which correspond to Legendre's three normal integrals. That is, the integral in eq. (2) may be finally reduced to the form

$$\int P/(Qy)\, dx = R_0(x)I_0 + R_2(x)I_1 + R_1(x)H_1 + R_3 \qquad (9)$$

where the $R_i(x)$ are rational functions in x. The recurrences (5) - (8) are complicated in nature and can lead to intractable intermediate expression swell. However, these are one-parameter relations, in contrast to the multi-parameter relations in Carlson's algorithm (Ref. 2). Therefore, this scheme may be a more appropriate candidate for mechanization than recurrences on Carlson's R-functions. On the other hand, Carlson's representation poses certain significant advantages that will be apparent in subsequent discussions.

Before any discussion of further reduction of eq. (9), we should indicate the existence of an alternative route (Ref. 3) to get to this equation from (2). Suppose we can perform a square-free decomposition on $Q(x)$ resulting in

$$Q(x) = (x-d_1)^{n_1}\ldots(x-d_j)^{n_j}(x-\alpha_1)^{m_1} \ldots(x-\alpha_i)^{m_i} \qquad (10)$$

where the α's are roots of y^2 and all roots are distinct. Furthermore, let

$$Q_1(x) = (x-\gamma_1)\ldots(x-\gamma_i) , \text{ and}$$

$$Q_2(x) = Q(x)/Q_1(x)$$

Then the second integral in eq. (2) may be expressed as

$$\int P_2/(Qy)dx = \frac{h(x)}{Q_2(x)}\, y + \int \frac{g(x)}{y}\, dx + \sum_{i=1}^{r} D_i \int \frac{dx}{(x-\gamma_i)y} \qquad (11)$$

here h and g are polynomials with undetermined coefficients, of degrees deg(Q)-r-1 and deg(y^2)-2 , respectively, deg(Q) being the degree of Q, etc. Differentiate eq. (11) and multiply through by Qy, we have an equation involving polynomials whose coefficients can be equated for like powers in x, thereby yielding a linear system which can be solved for the undetermined coefficients. At present this alternative is not considered viable because the matrix corresponding to the linear system is dense and quite sizable. Until a scheme is found to simplify this matrix (for example, decomposing into triangular or band matrices), this approach cannot be considered competitive.

Returning to eq. (9), we see four alternatives to proceed from here, in reducing I_0, I_1, and H_1 to canonical forms. They are the three classical approaches due to Legendre, Weirstrass, and Jacobi, and the recent one due to Carlson. The most widely known reduction procedure is due to Legendre. To illustrate this, we consider the simplest case, viz.,

y^2 is a cubic having three real roots, say $\alpha_1<\alpha_2<\alpha_3$. We also assume, without loss of generality, that y^2 is monic. To reduce I_0, etc. to the three canonical integrals of Legendre, i.e., $F(\phi,k)$, $E(\phi,k)$, and $\Pi(\phi, \alpha^2,k)$, it is needed first the following transformations (Ref. 6)

$$x = \frac{a \sin^2\phi+b}{c \sin^2\phi+d}, \qquad 0\leq\phi\leq\pi/2$$

$$dx = \frac{2D \sin\phi\cos\phi}{(c \sin^2\phi+d)^2} d\phi, \tag{12}$$

$$y = G\frac{\sin\phi\cos\phi \ \Delta}{(c \sin^2\phi+d)^2}, \quad \Delta^2 = 1-k^2 \sin^2\phi$$

Here the values of a, b, c, d, k, D=ad-bc, and G are given parameters determined by the integration intervals. For example, for the integration interval $x\geq\alpha_3$, we have

$$a = -\alpha_2, \ b = \alpha_3, \ c = -1, \ d = 1,$$
$$D = \alpha_3,-\alpha_2, \ k^2 = (\alpha_2-\alpha_1)/(\alpha_3-\alpha_1),$$
$$G = (\alpha_3-\alpha_2)\sqrt{\alpha_3-\alpha_1}$$

We then may express I_0, I_1, and H_1 in terms of Legendre's normal integrals as follows

$$I_0(x) = \frac{2D}{G}F(\phi,k) + \text{const.} \tag{13}$$

$$I_1(x) = \frac{2D}{G}\{\alpha_3F(\phi,k)-(\alpha_3-\alpha_1)E(\phi,k) + (\alpha^3-\alpha_1)\Delta\tan\phi\}+ \text{const.} \tag{14}$$

$$H_1(x) = \frac{2D}{G(a-\gamma_1c)}\{cF(\phi,k) + \frac{D}{b-\gamma_1d}\Pi(\phi, \frac{a-\gamma_1c}{b-\gamma_1d})\} + \text{const.} \tag{15}$$

For the integration variable lying in the open or closed intervals (α_1,α_2), (α_2,α_3), and $(0,\alpha_1)$ these parameters will be different. In the case y^2 has one real and two complex roots, the transformation eq. (12) and the corresponding parameters will be entirely different.

Thus the case of y^2 being cubic may be summarized as follows. There are two sets of transformations depending on whether all roots are real. For each set it is necessary to distinguish four sub-intervals making a total of eight cases. The situation of y^2 being quartic requires three sets of transformations (4 real roots, 4 complex roots, and 2 real and 2 complex). All in all, it appears to us that Legendre's canonical reduction would require a fair amount of storage of auxilliary information. This makes it unattractive as a candidate for a mechanical reduction procedure. As mentioned in a previous report (Ref. 2), in Carlson's approach, these distinct cases are simplified through permutation symmetry.

The second alternative we consider is a reduction to Weirstrass' normal integrals. Again let us first discuss the simplest case, viz., $y^2 = a_3x^3+a_2x^2+a_1x+a_0$.

The substitution

$$x = (\frac{4}{a_3})^{1/3}t-\frac{a_2}{3a_3}$$

$$g_2 = (\frac{a_2^2}{3a_3} - a_1)(\frac{4}{a_3})^{1/3}$$

$$g_3 = \frac{a_1a_2}{3a_a} - \frac{2a_2^3}{27a_3^2} - a_0$$

would enable eq. (2) to be expressible in terms of Weirstrass' p-function $p(u)$, viz.,

$$I_0 = \int_\infty^{p(u)} \frac{dt}{(4t^3-g_2t-g_3)^{1/2}} \tag{16}$$

$$I_1 = \int_{p(u_0)}^{p(u)} \frac{t \ dt}{(4t^3-g_2t-g_3)^{1/2}} \tag{17}$$

$$H_1 = \int_\infty^{p(u)} \frac{dt}{(t-\gamma_1)(4t^3-g_2t-g_3)^{1/2}} \tag{18}$$

For y^2 being a quartic similar transformations may be applied to express I_0, I_1, and H_1 in similar terms. From practical considerations, we have deemed the Weirstrass' reduction procedure not appropriate as a candidate for symbolic integration, for two major reasons. First, the Weirstrass' p-functions or related functions are relatively unknown to most scientists and engineers. Second, the Weirstrass' functions arose as instances of doubly-periodic functions and as such were only related to the present problem as its inverse. As corrolary to this, there has not been much investigation on the computation of the inverse p-functions, and the unavailability of both numerical algorithms and software is considered a deterrent to the candidacy of Weirstrass' method for symbolic integration.

A third alternative in the canonical reduction involves the use of Jacobi's elliptic

functions. This is the major method employed in the substantial tabulation of Byrd and Friedman (Ref. 4). To apply this approach we may start with eq. (9) or go back to eq. (2). If we take the former, the situation would be very similar to Legendre's case, where we need to distinguish various situations according to roots of y^2 and to different integration intervals. For example, for the case mentioned above, where $y^2 = (x-\alpha_1)(x-\alpha_2)(x-\alpha_3)$ and the integration interval $x \geq \alpha_3 > \alpha_2 > \alpha_1$, we have

$$sn^2\phi = \frac{x-\alpha_3}{x-\alpha_2}\ ,\quad k^2 = \frac{\alpha_2-\alpha_1}{\alpha_3-\alpha_1}\ ,\quad g = \left(\frac{4}{\alpha_3-\alpha_1}\right)^{1/2}$$

$$I_0 = g\ sn^{-1}(\sin\phi, k) \tag{19}$$

$$I_1 = \alpha_3 g \int_0^u \frac{1-\eta sn^2 u}{(1-sn^2 u)}\, du \tag{20}$$

$$H_1 = \frac{g}{(\alpha_3-\gamma_1)} \int_0^u \frac{cn^2 u\ du}{(1-\beta^2 sn^2 u)} \tag{21}$$

where $\beta^2 = (\alpha_2-\gamma_1)/(\alpha_3-\gamma_1)$, $\eta = \alpha_2/\alpha_3$

These Jacobian integrals in turn can be related to the Legendre integrals. Expressed in this form, the Jacobi approach is even less attractive than Legendre's as a candidate for symbolic integration because the results are given in terms of inverse functions while the same kind of storage of auxilliary information is needed as in Legendre's case. In other words, while there is no savings on computing and storage as far as reduction goes, the answer is given in even more complicated and less tractable forms.

However, the above may be an unfair assessment of the potential of Jacobi's method, because it is possible to start with eq. (2) and perform recurrence on the Jacobi intergals directly, instead of I_k and H_k . In fact, this is exactly the route taken by Byrd and Friedman. To do so, we consider again the second integral in eq. (2). A judicious choice of substitution of the form $sn^2(u)=r(x^2)$, with r being a simple rational function in x^2 , would transform the integral to one involving rational functions of sn(u), which in turn can be decomposed into partial fractions. This process then reduces most integrals into combinations of twelve Jacobian integrals of the form

$$J_m = \int [J_a(u)]^m\, du \tag{22}$$

where $J_a(u)$ is one of the twelve Jacobian functions, viz., sn, ns, cn, nc, dn, nd, tn, cs, ds, sd, cd, and dc. Each kind of J_m satisfies a recurrence relation which may be applied to reduce a given integral into combinations involving the Jacobian functions, elementary functions and as few as two inverse Jacobian functions. The last two can be chosen as u and E(am u, k) which would correspond to Legendre's $F(\phi,k)$ and $E(\phi,k)$. As a simple example we may cite the case for sn(u), where we have,

$$A_m = \int sn^m(u)\, du \tag{23}$$

$$A_0 = u$$

$$A_1 = \frac{1}{k}\log[(dn(u)-kcn(u))]$$

$$A_2 = [u-E(amu,k)]/k^2$$

$$A_3 = [kcn(u)dn(u)+(1+k^2)\log(dn(u)-kcn(u))]/(2k^3)$$

$$(2m+1)k^2 A_{2m+2} = sn^{2m-1}(u)cn(u)dn(u)+2m(1+k^2)A_2 + (1-2m)A_{2m-2} \tag{24}$$

$$2(m+1)k^2 A_{2m+3} = sn^{2m}ucn(u)dn(u)+(2m+1)(1+k^2)A_{2m+} -2mA_{2m-1} \tag{25}$$

It may be noted paranthetically that the A_{2m+1}'s reduce to a combination of Jacobi and elementary functions where as A_{2m}'s reduce to a combination including the inver functions also. This property is true with all J_m's, and means that the odd-degree integrals are elementary while the even-degr integrals are transcendental (though it may not be apparent in the current context.) T an algebraic function may then be integrate by the following process:

given $R(x)=[4\beta^2+(2x-a-b)^2]^{1/2}[(x-a)(x-b)]$

let $sn^2u = [4\beta^2+(a-b)^2]/[4\beta^2+(2x-a-b)^2]$

$k^2 = 4\beta^2/[4\beta^2+(a-b)^2]$

$g = [4\beta^2+(a-b)^2]^{-1/2}$

$\sin\phi = sn\ u_1 = 2\beta[4\beta^2+(2y-a-b)^2]^{-1/2}/$

$y \geq a > b > 0$

then $\int_y^\infty R(x)dx = g^3 \int_0^{u_1} sn^2 u\, du$ (26)

As a candidate for automation, Jacobi's approach is saddled with three fundamental difficulties. First, the choice $sn^2u=r(x^2)$ mentioned above is by no means mechanical. In Byrd and Friedman, this choice is taken judiciously, requiring expert human judgement. Second, even if the first hurdle can be overcome, this approach still requires substantial storage, such as the J_m's, the numerous parameters for different integration intervals, and the various cases of roots of y^2. Third, the procedure is not characterized by any closure property in the sense that a finite list of basis functions is sufficient to represent all members of the class $R(x,y)$, with deg (y^2) less than five. In particular the list of J_m's is not sufficient to represent all such integrals of algebraic functions. To supplement the J_m's, Byrd andFriedman have to include a substantial list of additional integrals that involve various combinations of the elliptic functions.

In summary, three classical methods have been examined as potential candidates for symbolic integration. Each method presents serious difficulties. At this point, one compromise appears to be a combination of Carlson's approach described in a previous report (Ref. 2) and the recurrence discussed in the beginning of this section. Another compromise is to modularize this entire process into two stages. The first stage basically simulates the steps described by eqs. (1)-(9) and therefore is independent of which particular method or representation one chooses. The second stage may consist of several modules to simulate the various methods described in this presentation. For example, one may choose to express the right hand side of eq. (9) in terms of Carlson's R-functions or Legendre's integrals.

At some point we decided that the two-stage compromise had more appeal to us, mainly for the reason of its flexibility in adaptation to the various representations. We therefore have first constructed MACSYMA programs for the simulation of the steps in eqs. (1)-(9). These programs are basically constituted of several modules. The first is a "front-end" supervisory program called ELLIPINT(R,F,X,Y), which takes $R(x,y)$ as input and sets it to the form given by eq. (2). Here F is the polynomial equal to y^2. Then two subprograms named ELLIP1(REMM,ROOT,M,F,X) and ELLIP2(QUO,F,X) prepare the two integrals of the right hand side to be expressed in terms of I_k and H_k. Finally, two subprograms JJ(N,F,X) and H(M,ROOT,F,X) implement the recurrence relations for I_k and H_k, respectively. The end result of this sequence of applications is to be the right hand side of Eq. (9). In the oral presentation, we shall describe our experience with these programs.

ACKNOWLEDGEMENT

This investigation has been supported by an Office of Naval Research Grant #NR004-67, and a NASA contract No. NAS7-100.

REFERENCES

1. Moses, J., "Symbolic Integration--The Stormy Decade", Comm.ACM, Vol. 14, pp. 548-560. (1971)

2. Ng, E. W. , "Symbolic Integration of A Class of Algebraic Functions", Jet Propulsion Laboratory Technical Memorandum No. 33-713. (1974)

3. Grobner, W. and N. Hofreiter, "Integraltafel: Unbestimmte Integrale", Springer-Verlag: Wien. (1965)

4. Byrd, P. and M.D. Friedman, "Handbook of Elliptic Integrals for Engineers & Scientists", Springer-Verlag: N.Y. (1971)

Symbolic/Numeric Algorithms for Partial Differential Equations

Dana L. Ulery
and
Hatem M. Khalil
University of Delaware

ABSTRACT

Symbolic computation has been used to generate classes of numeric algorithms for solving linear partial differential equations. A two-phase procedure is described for automatically producing multiparameter families of difference approximations to the heat operator. In the first phase, a truncated series expansion is used to form the appropriate difference operator. The coefficients of the operator, functions of free parameters, are determined in the second phase. When the region is rectilinear, Vandermonde matrices are constructed which permit this phase to be computed numerically. When the region is irregular, a symbolic approach appears to be more efficient.

KEY WORDS AND PHRASES: symbolic mathematical computation, partial differential equations, finite differences.

CR CATEGORIES: 5.14, 5.17, 5.7

1. INTRODUCTION

The combined use of numeric and symbolic techniques is proving quite valuable in defining consistent families of finite difference operators for approximating linear partial differential equations. These families depend on more than one parameter, and in fact the number of free parameters can grow to be quite large. Such families identify classes of algorithms and permit systematic complexity studies in terms of the parameters.

2. THE HEAT OPERATOR IN A RECTILINEAR REGION

The method used to generate multi-parameter families of difference operators will be demonstrated for the heat operator in one and two-dimensions. As the simplest case, consider the one-dimensional heat operator

$$\wedge[u(x,t)] = u_t - u_{xx} = 0$$

in a rectilinear region R of the (x,t) space with boundary S, subject to initial conditions

$$u(x,0) = f(x)$$

with x in R and Dirichlet boundary conditions

$$u(x,t) = g(x,t) \text{ with } x \text{ on } S.$$

To solve this problem numerically, we cover R with a uniform lattice of discrete points and approximate $\wedge[u(x,t)]$ using a two-level stencil of six points

$$x_a = x + J_a h$$

$$t_\alpha = t + K_\alpha k$$

where

$$J_a = a - 2 \text{ for } 1 \le a \le 3$$

$$K_\alpha = \alpha - 1.5 \text{ for } \alpha = 1, 2$$

$$h = \Delta x > 0.$$

$$k = \Delta t > 0.$$

We then approximate $\wedge[u(x,t)]$ by the difference operator

$$L[u(x,t)] = \sum_{\alpha=1}^{2} \sum_{a=1}^{3} A_{\alpha,a}\, u(x+J_a h,\ t+K_\alpha k)$$

where $A_{\alpha,a}$ are constant coefficients whose values must be determined. We base our technique on the method of undetermined coefficients. Expand $u(x+J_a h,\ t+K_\alpha k)$ about the central points (x,t) as a Taylor's series in powers of $J_a h$ and $K_\alpha k$ [6] to obtain

$$u(x+J_a h,\ t+K_\alpha k) = \sum_{p=0}^{\infty} \sum_{q=0}^{\infty} J_a{}^p K_\alpha{}^q \frac{h^p k^q}{p!q!} \frac{\partial^{p+q} u(x,t)}{\partial x^p\, \partial t^q}.$$

Substituting expanded values of u into the difference operator, we obtain

$$L[u(x,t)] = \sum_{p=0}^{\infty} \sum_{q=0}^{\infty} B_{p,q} \frac{h^p k^q}{p!q!} \frac{\partial^{p+q} u(x,t)}{\partial x^p\, \partial t^q}$$

where $B_{p,q} = \sum_{\alpha} \sum_{a} A_{\alpha,a} J_a^p K_\alpha^q$.

To uniquely define the coefficients and minimize the order of the error, we select terms defined by $0 \le p \le 2$, $0 \le q \le 1$, so that

$$L[u(x,t)] = B_{00}u + B_{10}hu_x + B_{20}\frac{h^2}{2}u_{xx}$$

$$+B_{01}ku_t + B_{11}hku_xu_t + B_{21}\frac{h^2k}{2}u_{xx}u_t + O(h^3) + O(k^2).$$

To assure that the difference operator includes the explicit terms of the heat operator, we demand that

$$B_{20} = -2h^{-2} \text{ and } B_{01} = -k^{-1} .$$

Moreover, we insist that

$$B_{00} = B_{10} = 0$$

to insure that the error term is at least $O(hk)$. If we set $r = k/h^2$ and make the typical assumption that $r = O(1)$, then the operator will be at worst $O(h^3)$.

The remaining coefficients, B_{11} and B_{21}, are arbitrary generation parameters used to define this family of difference operators.

We are now in a position to determine the coefficients $A_{\alpha,a}$. We take advantage of the fact that J_a^p and K_α^q in the six linear equations defining $B_{p,q}$ for $0 \le p \le 2$, $0 \le q \le 1$, define the Vandermonde matrices

$$V_3 = [V_{a,p+1} = J_a^p] = \begin{bmatrix} 1 & -1 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

and $$V_2 = [V_{\alpha,q+1} = K_\alpha^q] = \begin{bmatrix} 1 & -1/2 \\ 1 & +1/2 \end{bmatrix} ,$$

whose inverses are readily found to be

$$Z_3 = [Z_{p+1,a}] = \begin{bmatrix} 0 & 1 & 0 \\ -1/2 & 0 & 1/2 \\ 1/2 & -1 & 1/2 \end{bmatrix} .$$

and $$Z_2 = [Z_{q+1,\alpha}] = \begin{bmatrix} 1/2 & 1/2 \\ -1 & 1 \end{bmatrix} .$$

Inverting the six linear equations for $B_{p,q}$, we obtain

$$A_{\alpha,a} = \sum_{p=0}^{2} \sum_{q=0}^{1} B_{p,q} Z_{p+1,a} Z_{q+1,\alpha} .$$

Thus, the coefficients $A_{\alpha,a}$ depend on the two generation parameters, B_{11} and B_{21}, as well as h and k.

It is important to observe that the Vandermonde formalism, developed by Khalil and Giese [2], is the essential feature that permits this technique to be carried to completion numerically. In the first place, special algorithms for the efficient inversion of Vandermonde matrices exist. Secondly, the ability to split the problem into component matrices permits inversions of matrices of much lower order. For example, consider the two-dimensional case. An immediate extension of the method yields eighteen linear equations

$$B_{s,p,q} = \sum_{\alpha=1}^{2} \sum_{a=1}^{3} \sum_{b=1}^{3} A_{a,b}^{\alpha} J_a^s J_b^p K_\alpha^q ,$$

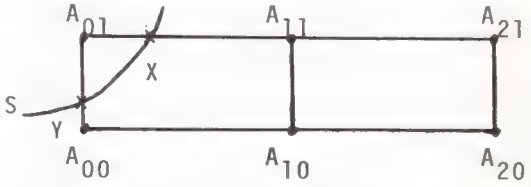
for $0 \le s, p \le 2$, $0 \le q \le 1$. To solve this system directly requires that we invert an 18 x 18 matrix. Using the method suggested above, the solution is given as

$$A_{a,b}^{\alpha} = \sum_{s=0}^{2} \sum_{p=0}^{2} \sum_{q=0}^{1} B_{s,p,q} Z_{s+1,a} Z_{p+1,b} Z_{q+1,\alpha} ,$$

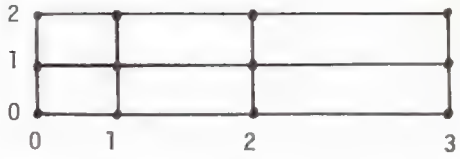
and requires the inversion of two matrices each of order three, and one matrix of order 2. Clearly, as the number of dimensions increases, the use of Vandermonde matrices becomes increasingly advantageous.

3. THE HEAT OPERATOR IN AN IRREGULAR REGION

Let us now consider this same problem in an arbitrary region. The Vandermonde formalism is dependent upon a uniform stencil. To handle points along an irregular boundary, we must somehow modify the basic stencil. Consider the case where a single point of the basic stencil falls outside the domain, shown in the diagram below:



where {X,Y} denote the points at which the boundary S cuts the stencil. One possible modification is to let X and Y define new lines on the stencil, so that we have a stencil of the form



in which twelve coefficients are specified [1]. We can then write

$$B_{p,q} = \sum_{\alpha=1}^{3} \sum_{\alpha=1}^{4} A_{\alpha,a} J_a^p K_\alpha^q$$

for $0 \le p \le 3$, $0 \le q \le 2$.

We again demand that $B_{20} = -2h^{-2}$, $B_{01} = -k^{-1}$, $B_{00} = B_{10} = 0$, leaving eight generation parameters. To solve for the coefficients requires the numeric inversion of a fourth and third order Vandermonde matrix. In the interior of the region we are still using a six-point stencil, and therefore would like the modified stencil to be as compatable as possible. We might decide to restrict the modified stencil to six points as well, by forcing six coefficients to vanish. The algorithm for deciding which points to retain is not at all obvious, particularly as the number of points exterior to the domain and number of space dimensions increase. Let us assume for this example that we decide to force

$$A_{00} = A_{10} = A_{11} = A_{21} = A_{31} = A_{02} = 0,$$

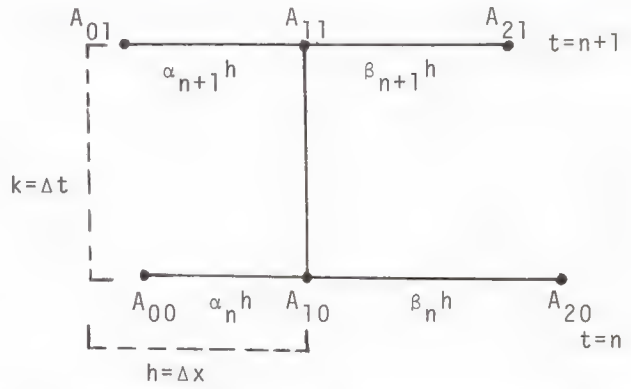
thus imposing six additional constraints on the original system, leaving again a two-parameter family of difference operators. Since the coefficients have been expressed in terms of eight generation parameters $B_{p,q}$,

$$A_{\alpha,a} = \sum_{p=0}^{3} \sum_{q=0}^{2} B_{p,q} Z_{p+1,a} Z_{q+1,\alpha},$$

we must solve a sixth order underdetermined homogeneous system to determine values for the two parameters which are free. We can assign arbitrary values to the free parameters and continue with a numeric solution, or solve the system symbolically. In any event, we have gained nothing in this instance by using the Vandermonde formalism, and have in fact made the task more difficult than it would have been taking the direct approach. Had we looked at the two-dimensional case with a single exterior stencil point, we would have had to finally solve a fourteenth order under-determined homogeneous system. Using a numeric approach, the increase in complexity for a specific case is not significant; however, each specific case must be handled individually, and the number of cases to be considered grows rapidly with the number of spatial coordinates. This results in significant increase of complexity in the generation of families.

- It is therefore worthwhile to consider a completely symbolic approach to this problem. We will retain the technique of undetermined coefficients to automatically generate consistent multiparameter families, but forego the Vandermonde formalism in favor of a more direct approach. As we shall see, the essence of the formalism is not totally lost, and can be used to advantage in the symbolic method as well as in the numeric.

We choose a lattice of variable geometry and introduce explicit spatial displacement parameters (SDP) to achieve nonuniformity of step size along the spatial axes. This device allows us to systematically generate multiparameter families of difference operators with the same number of grid points in the interior and at the boundaries, thus minimizing complexity. Again, we approximate $\wedge[u(x,t)]$ with a two-level stencil of six points, but now demand that the four end points be adjustable so that they can be positioned to lie directly on the boundary S. Let α_n, β_n, α_{n+1}, β_{n+1} denote the left and right SDP at the lower and upper time levels respectively. The stencil is shown below:



The difference operator associated with this stencil is given by

$$L[u(x,t)] = A_{00}u(x-\alpha_n h, t-1/2k) + A_{10}u(x, t-1/2k)$$
$$+A_{20}u(x+\beta_n h, t-1/2k) + A_{01}u(x-\alpha_{n+1}h, t+1/2k)$$
$$+A_{11}u(x, t+1/2k) + A_{21}u(x+\beta_{n+1}h, t+1/2k),$$

where $0 < \alpha, \beta \leq 1$.

To compute the coefficients $A_{\alpha,a}$ we proceed as before, expanding $L[u(x,t)]$ about the central point (x,t) as a Taylor's series, and selecting (p,q) pairs to include $\wedge[u(x,t)]$ and forcing the u and u_x terms to vanish. The coefficients are then uniquely determined by the system of linear symbolic equations:

$$Ax = f$$

where

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ -\alpha_n & 0 & \beta_n & -\alpha_{n+1} & 0 & \beta_{n+1} \\ \alpha_n^2 & 0 & \beta_n^2 & \alpha_{n+1} & 0 & \beta_{n+1}^2 \\ -1/2 & -1/2 & -1/2 & 1/2 & 1/2 & 1/2 \\ 1/2\alpha_n & 0 & -1/2\beta_n & -1/2\alpha_{n+1} & 0 & 1/2\beta_{n+1} \\ -1/2\alpha_n^2 & 0 & -1/2\beta_n^2 & 1/2\alpha_{n+1}^2 & 0 & 1/2\beta_{n+1}^2 \end{bmatrix}$$

$$x = (A_{00}A_{10}A_{20}A_{01}A_{11}A_{21})^T,$$

$$f = (\,0\ 0\ -2h^{-2}\ k^{-1}\ \xi\ \eta)^T,$$

and $\xi \equiv B_{11}$ and $\eta \equiv B_{21}$ are two generation

parameters.

This system was solved by MACSYMA, yielding explicit expressions for the $A_{\alpha,a}$ in terms of $\alpha_n, \alpha_{n+1}, \beta_n, \beta_{n+1}, \xi, \eta$, h and k. These coefficients then define the desired two generation parameter (GP), four SDP family of consistent difference operators. This family defines a class of algorithms for solving the one-dimensional heat operator in an arbitrary region with Dirichlet boundary conditions. The complexity of this family can be studied both analytically and numerically, in terms of the parameters.

This sixth order system of linear equations pushes present symbol manipulating systems close to their limits of capabilities. MACSYMA was unable to solve a similar tenth order system. It is, therefore, necessary to seek ways to reduce the order of the system to be solved. To begin with, we note that the matrix A can be partitioned as

$$A = \left[\begin{array}{c|c} V_1 & V_2 \\ \hline -1/2V_1 & V_2 \end{array}\right]$$

where V_1 and V_2 are Vandermonde matrices of order three. Using this property, MACSYMA can solve Ax = f by simply inverting two third order matrices, since

$$A^{-1} = \left[\begin{array}{c|c} 1/2V_1^{-1} & -V_1^{-1} \\ \hline 1/2V_2^{-1} & V_2^{-1} \end{array}\right].$$

Multiparameter families of difference operators for approximating higher dimension heat equations with Dirichlet boundary conditions in arbitrary regions can be constructed from this basic stencil. The construction involves simple algebraic combination of operators, and requires no additional symbolic computation. The coefficients of these higher dimension difference operators are obtained explicitly this way [3].

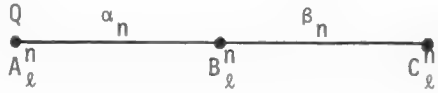
4. MIXED BOUNDARY CONDITIONS

Let us finally consider the problem with mixed boundary conditions

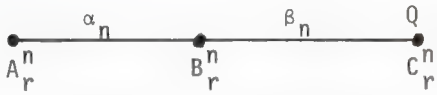
$$c(t)u(x,t)+d(t)u_x(x,t)=g(x,t) \text{ with } x \text{ on } S.$$

To maintain a difference operator of the same order both on the boundary and in the interior, we approximate discrete boundary points with a three-point stencil [5].

If Q is a point on the left boundary S at time n, the stencil assumes the form



where A_ℓ^n, B_ℓ^n, C_ℓ^n are constant coefficients; α_n, β_n are SDP; and A_ℓ^n is the coefficient associated with $Q \varepsilon S$. Similarly, at the right boundary we use an analogous stencil



such that C_r^n is the coefficient associated with a point on the boundary. The associated difference operators are

$$L_d^s[u]=A_d^s\, u(x-\alpha_s h, t+\gamma_s k) + B_d^s\, u(x,t+\gamma_s k)$$
$$+C_d^s\, u(x+\beta_s h, t+\gamma_s k),$$

where $s = n, n+1$

$d = \ell, r$

$\gamma_s = s-(n+1/2)$.

We expand the operators $L_\ell^s[u]$ in a Taylor's series about the leftmost point $(x-\alpha_x h, t+\gamma_s k)$ and select terms which include the first derivative and assure an error $O(h^3)$. This leads to the system of linear symbolic equations

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & \alpha_s & \alpha_s+\beta_s \\ 0 & 1/2\alpha_s^2 & 1/2(\alpha_s+\beta_s)^2 \end{bmatrix} \begin{bmatrix} A_\ell^s \\ B_\ell^s \\ C_\ell^s \end{bmatrix} = \begin{bmatrix} 0 \\ h^{-1} \\ 0 \end{bmatrix}$$

which uniquely determines the coefficients as functions of the SDP. The coefficients of $L_r^s[u]$ are similarly determined.

Substitution of these finite difference approximations to the first derivative in the boundary conditions results in expressions for points on the boundary given explicitly in terms of interior stencil points and SDP. These expressions are in turn substituted into the basic six-point operator L[u(x,t)], to form a general two-level, six-point operator $L_G[u(x,t)]$ appropriate without modification both in the interior of the region and along the boundary. The coefficients of this general operator are functions of the two GP(ξ and η); the functions c(t), d(t), and g(x,t) appearing in the boundary conditions; four SDP; and h and k--a total of eleven variables.

The totality of difference equations described by $L_G[u(x,t)]$ can be expressed in matrix form

$$M_{n+1}U^{n+1} = M_n U^n + Z^n$$

where M_{n+1} and M_n are square tridiagonal matrices, U^{n+1} and U^n are column vectors of solution points at time n+1 and n, respectively, and Z^n is a column vector

subdivided into 2^n subsimplexes, those subsimplexes into 2^n sub-subsimplexes, etc. to m levels, the total number of mesh points is $\binom{n + 2^m}{n}$ and this is precisely the number of terms in a polynomial of degree 2^m in n variables.

We cannot delve deeply into the algorithm here, we only discuss the construction and use of the polynomial interpolant over a simplex, since there both symbolic and numerical tools come into play.

We assume f is defined over an n-simplex and wish to construct a dth degree polynomial interpolant. For purposes of illustration, we consider n = 2 and d = 2. The simplex is then a triangle and six $\left[= \binom{n + d}{n} \right]$ points are required to determine the polynomial since that is the number of monomials of degree ≤ 2 in two independent physical variables. An efficient method of constructing this approximation is based on a scheme of Silvester [5] which uses a barycentric coordinate system over the simplex and produces a polynomial in three [= n+1] barycentric variables. The method precomputes certain basic polynomials over a general simplex. The jth polynomial is constructed to evaluate to 1 at the jth of six selected points and to 0 at the others. A simple linear combination of these basic polynomials then yields the interpolant for a given specific simplex. The six points used are, in barycentric coordinates,

$$\langle c_0/d \ , \ c_1/d \ , \ c_2/d \rangle$$

where $c = \langle c_0, c_1, c_2 \rangle$ is a "composition" of 2[= d] into 3[= n+1] parts allowing zero parts, that is, the c_i are integers such that

$$0 \leq c_i \leq 2 \quad \text{and} \quad d = \sum_{0 \leq i \leq 2} c_i = 2 \ .$$

[As mentioned earlier, when $d = 2^m$, these points are vertices of the sub$\cdots$ subcells to level m, hence Silvester's scheme merges nicely with our "adaptive" algorithm.] Figure 1 shows the triangle, its points of evaluation for a quadratic polynomial given in barycentric coordinates and the basic polynomials associated with those points given in barycentric variables.

4. SYMBOLIC COMPUTATION

Let $p = \langle c_0/d \ , \ c_1/d \ , \ \ldots, \ c_n/d \rangle$ be one of the points at which a basic polynomial is to be calculated, that is

$$\sum_{0 \leq i \leq n} \frac{c_i}{d} = 1$$

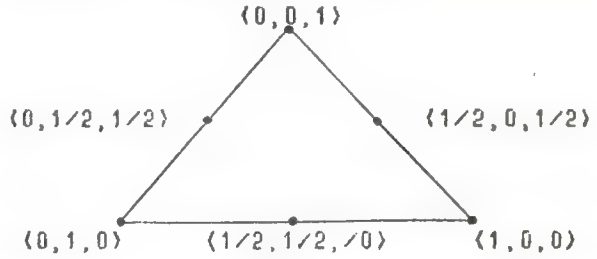
for integral c_i . The polynomial is

$$Q = \prod_{0 \leq i \leq n} R_i(z_i)$$

where

$$R_i(z_i) = \frac{1}{c_i!} \prod_{0 \leq k \leq c_i} d \cdot z_i - k \ .$$

Each R_i is a polynomial in the single distinct variable z_i and Q is a polynomial in the n + 1 variables z_i, $0 \leq i \leq n$. It is straightforward to verify that Q equals 1 at the point p and equals 0 at all other evaluation points.



$$Q_{\langle 1,0,0 \rangle} = z_0(2z_0 - 1) \qquad Q_{\langle 1/2,1/2,0 \rangle} = 4z_0z_1$$
$$Q_{\langle 0,1,0 \rangle} = z_1(2z_1 - 1) \qquad Q_{\langle 1/2,0,1/2 \rangle} = 4z_0z_2$$
$$Q_{\langle 0,0,1 \rangle} = z_2(2z_2 - 1) \qquad Q_{\langle 0,1/2,1/2 \rangle} = 4z_1z_2$$

Figure 1: Polynomial Bases for n=2, d=2 .

Computer calculation of Q is also straightforward provided we can do polynomial arithmetic easily. The algorithm is expressed as a procedure in the Madcap language in Fig. 2. This program makes use of a "space" of polynomials in which polynomials are represented by tuples of coefficients and can be manipulated with various operators. This program uses the

```
«
 c: composition
 n ← #c-1; d ← Σ0≤i≤nci @ POLYNOMIAL
 z ← variable↑POLYNOMIAL
 Q ← Π[for 0≤i≤n: π(for 0≤k<ci: d·z-k)·1/ci!]
»
```

Figure 2: Calculation of Interpolating Polynomial.

dot multiply (·) and subtract (-) to form the linear polynomials and for constant multiplication, the small product (π) to form the univariate R's (the innermost product), and the big product (Π) to form

the multivariate Q.

Now let $p_0, p_1, \ldots, p_{v-1}$ be the $\binom{n+d}{n}$ evaluation points and let $Q_0, Q_1, \ldots, Q_{v-1}$ be the corresponding basis polynomials (independent of f). The linear combination

$$A = A(z_0, \ldots, z_n) =$$

$$Q_0 \cdot f(p_0) + Q_1 \cdot f(p_1) + \cdots + Q_{v-1} \cdot f(p_{v-1})$$

is then the polynomial approximation to f over a specific simplex in barycentric variables. [This formula shows the need for a plus (+) operator in our polynomial space. Other useful operators include subscript for picking off coefficients and juxtaposition for evaluating a polynomial at a point in n-space.] This polynomial can now be evaluated without repeating the time-consuming computation needed to construct it.

5. NUMERIC COMPUTATION

Numeric computation in the form of matrix arithmetic arises naturally in the evaluation of the polynomial interpolant, essentially for the conversion between points given in physical coordinates to points given in barycentric coordinates. Let (x_0,y_0), (x_1,y_1), (x_2,y_2) be vertices of a triangle given in physical coordinates, and $p = (x,y)$ be an arbitrary point in 2space. The barycentric coordinates $\langle b_0,b_1,b_2 \rangle$ of p relative to the triangle may be calculated from the relations

$$b_0x_0 + b_1x_1 + b_2x_2 = x$$

$$b_0y_0 + b_1y_1 + b_2y_2 = y$$

and

$$b_0 + b_1 + b_2 = 1$$

In matrix form, this is

$$\langle b_0, b_1, b_2 \rangle = \langle x, y, 1 \rangle \cdot T^{-1}$$

where

$$T = \begin{pmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{pmatrix}$$

Thus, to evaluate the polynomial A at one of many points p, one first calculates the barycentric coordinates of p using matrix operations, and then substitutes these as values of the barycentric variables of A.

A program for calculating the point $B = \langle b_0, b_1, \ldots, b_n \rangle$ given a physical point $X = \langle x_0, x_1, \ldots, x_{n-1} \rangle$ and the precomputed inverse to T appears in Fig. 3. This and other matrix

```
«
X: point; T.inverse: @MATRIX
v ← ((x_i: 0≤i<n; 1)) @ MATRIX
B ← v · T.inverse
»
```

Figure 3: Conversion from Physical to Barycentric Coordinates.

manipulation programs make use of a space of matrices (including vectors) with representation via tuples of tuples and the operators of multiply (·) and invert (exponentiation to -1). Thus, numeric as well as symbolic manipulation is expressed with natural notation.

References

[1] Dahl, O. J., Myhrhaug, B., and Nygaard, K., *The SIMULA 67 Common Base Language*, Publication S-22, Norwegian Computing Center, Oslo, 1970.

[2] Kahaner, D. K. and Wells, M. B., "An Algorithm for n-dimensional Adaptive Quadrature Using Advanced Programming Techniques" (submitted for publication).

[3] Liskov, B. and Zilles, S., "Programming With Abstract Data Types," *Proceedings of ACM SIGPLAN Conference on Very High Level Languages*, SIGPLAN notices, Vol. 9, no. 4 (April 1974), pp. 5059.

[4] Milora, S. L. and Tester, J. W., *Geothermal Energy as a Source of Electric Power*, MIT Press, 1976.

[5] Silvester, P., "Symmetric Quadrature Formulae for Simplexes," *Mathematics of Computation*, Vol. 24, no. 109, (January 1970), pp. 95-100.

[6] Wells, M. B. and Cornwell, F. L., "A Data Type Encapsulation Scheme Utilizing Base Language Operators," *Proceedings of ACM Conference on Data: Abstraction, Definition, and Structure*, Salt Lake City, March, 1976.

Proceedings and Back Issues

1. Proceedings of EUROSAM 74
   (SIGSAM Bulletin No. 31, August 1974
   20 papers, 110 pages)

   held: August 1-2, 1974, Stockholm, Sweden
   editor: Richard Jenks, IBM
   price: $6.00 for ACM or SIGSAM members
   $8.00 for others

2. Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation
   (47 papers; 8 abstracts; 464 pages)

   held: March 23-25, 1971, Los Angeles, California
   editor: Stanley Petrick, IBM
   price: $15.00

3. Back Issues of the Bulletin $1.00 per copy

From: SIGSAM, c/o Association of Computing Machinery, Inc., P.O.Box 12105, Church Street Station, New York, N.Y. 10249

Only prepaid orders filled

APPLICATION FOR MEMBERSHIP

ASSOCIATION FOR COMPUTING MACHINERY
1133 AVENUE OF THE AMERICAS, NEW YORK, N. Y. 10036

Please enroll me as a member of the SPECIAL INTEREST GROUP ON SYMBOLIC AND ALGEBRAIC MANIPULATION

NAME (Please print or type) _____

POSITION _____

COMPANY _____

MAILING ADDRESS _____

CITY _____ STATE _____ ZIP _____

ANNUAL MEMBERSHIP DUES

☐ ACM MEMBER $ 5.00
Number _____

☐ STUDENT MEMBER $ 2.50
Number _____

☐ NON-ACM MEMBER $ 8.00 *

☐ SUBSCRIPTION $ 12.50

☐ PLEASE SEND INFORMATION ON ACM MEMBERSHIP

* Non-ACM Membership is open only to individuals whose major professional allegiance is in other than information processing.